Deep Learning

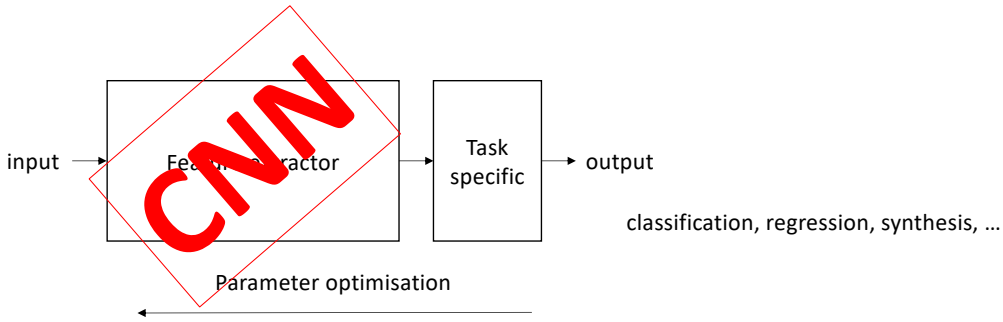
Bernhard Kainz

Motivation

- Deep learning is **popular** because it works (often).
  - Big promise: just collect enough data and label it, then you get a magic black-box predictor that can predict any correlations at the click of a button. (only supervised setting really works well)
- Deep learning and Big data = **big money** = highly competitive and sometimes poisonous working environment.
- Deep learning can be **dangerous**, e.g. deep fakes, adversarial attacks, etc.

Why deep learning is such a hot topic in today's tech landscape. At its core, the promise of deep learning is tantalizingly simple: collect enough data, label it, and voila -- you get what seems like a magic black-box predictor capable of astonishing tasks. But, let's add a caveat: this mainly holds true in supervised learning settings.
Now, it's crucial to understand that where deep learning and big data converge, big money follows. This has made the field not only highly competitive but also, unfortunately, sometimes a stressful and even toxic work environment.
But the story doesn't end there; deep learning isn't just roses and rainbows. It has a dark side. Technologies like deep fakes and the potential for adversarial attacks pose serious ethical and security concerns.
So, as we marvel at the capabilities of deep learning, it's crucial to also keep in mind these complexities—both the immense promise and the potential dangers -- so that we navigate this landscape responsibly.

Fundamental learning system



*CNN = convolutional neural network

The fundamental architecture of a deep learning system generally comprises three main components.
First up is the 'Feature Extractor.' In the context of image analysis, this often becomes a Convolutional Neural Network, or CNN. The feature extractor is responsible for capturing the hierarchical patterns in the data—edges, textures, and more complex shapes. This is what allows the network to 'understand' what it's looking at.
Next, we have the 'Task-Specific Head.' This component is tailored to the problem we're trying to solve. Whether it's image classification, object detection, or any other task, this part of the network is designed to take the rich features extracted by the CNN and map them to the specific task at hand.
The final component is 'Parameter Optimization.' This is where the magic happens, so to speak. Using algorithms like stochastic gradient descent, the model fine-tunes its parameters to minimize a loss function. This process can be applied to a variety of tasks—whether it's classification, regression, or even synthesis of new data points.

Success stories

Self driving cars: https://youtu.be/zRnSmw1i_DQ

Conversational AI: https://youtu.be/Xw-zxQSEzqo
https://youtu.be/jH-6-ZIgmKY https://chat.openai.com/

Deep fakes: https://youtu.be/gLoI9hAX9dw

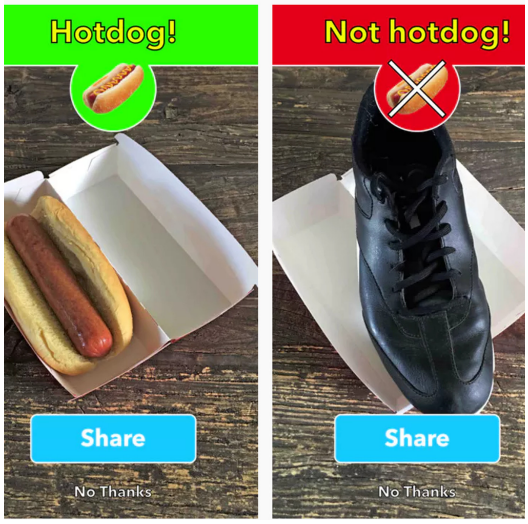
Neural rendering: https://www.matthewtancik.com/nerf

Image colourization: https://youtu.be/mUXpxxyThr8

Image captioning: https://youtu.be/8BFzu9m52sc

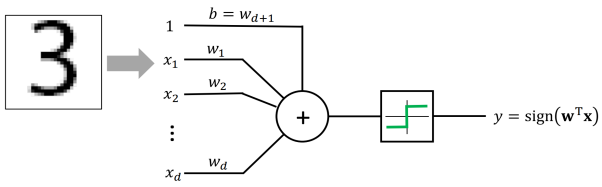
Automated diagnosis: http://ratchet.lucidifai.com/

Protein discovery: https://alphafold.ebi.ac.uk/



HBO and *Silicon Valley* *engadget.com*

Why did neural networks fail in image analysis?



Stack a 32x32x3 RGB image into a 3072x1 vector

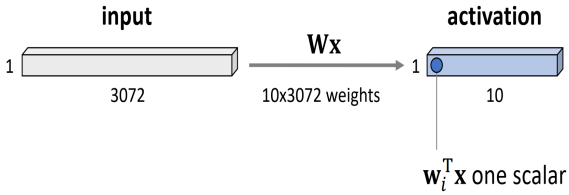


Figure: adapted from Fei Fei et al.

Traditional neural networks were ground-breaking for many applications, but when it came to image and text analysis, they had their drawbacks. One of the primary issues is dealing with the sheer complexity and size of image data. We're talking about a huge grid of pixels, and this can easily overwhelm a vanilla neural network.

Let's consider the way humans interpret images -- we naturally recognize hierarchies and patterns. Traditional neural networks, however, don't have an innate understanding of the spatial hierarchies that are present in image data.

Another critical challenge is translation invariance. For instance, if a traditional neural network is trained to recognize a cat in the bottom-left corner of an image, it may not easily recognize that same cat if it appears in the top-right corner. In essence, it lacks the flexibility to adapt to objects appearing in different locations within the image.

Lastly, overfitting becomes a significant concern with image data. The risk amplifies because of the high number of parameters involved in these models.

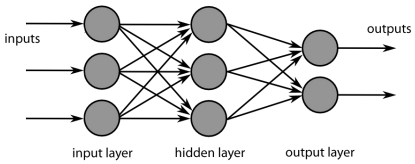
All these challenges underscore why the rise of CNNs was so crucial -- they were designed specifically to address these limitations. We'll delve deeper into CNNs shortly, but understanding these challenges really helps us appreciate why advancements in this field were so imperative.

Universal Approximator

- Let $\varphi(\cdot)$ be a non-constant, bounded and monotonically increasing function
- For any $\epsilon > 0$ and any continuous function defined on a compact subset of $\mathbb{R}^m$, there exists and integer N, real constants $v_i b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}$ where $i = 1, \dots, N$, such that

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \varphi(\boldsymbol{w}_i^T \boldsymbol{x} + b_i) \; with \; |F(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon$$

We can approximate *any* function with just one hidden layer with a sensible actitation function!
In practice ϵ very large and curse of dimensionally!
Solution: break up problem in many smaller problems (layers)



A mathematical underpinning of neural networks known as the Universal Approximation Theorem.

The theorem states that let $\varphi(\bullet)$ be a non-constant, bounded, and monotonically increasing function.
What this means is that φ is a sensible activation function, like the sigmoid or ReLU, that we often use in neural networks.

For any positive ϵ and any continuous function defined on a compact subset of $\mathbb{R}$^m, the theorem assures us that we can approximate this function to within an error of ϵ. Mathematically, this is represented as $F(\boldsymbol{x})$ being an approximation of $f(\boldsymbol{x})$, such that the absolute difference between them is less than ϵ.

In layman's terms, what this theorem tells us is that with just one hidden layer and a sensible activation function, we can approximate any function!
That's a big claim, right?

However, there are practical challenges.
Firstly, the error term ϵ can be very large in practice, making the approximation less useful.

Secondly, we encounter what's known as the 'curse of dimensionality,' where the computational and spatial complexity increases exponentially with the number of dimensions.

To tackle these issues, the typical solution is to break up the problem into many smaller problems, which is essentially what multiple layers in deep neural networks do.
This hierarchical decomposition allows us to learn complex functions more effectively.

The curse of dimensionality

So what is the curse of dimensionality?

Curse of dimensionality

As the number of features or dimensions grows,
the amount of data we need to generalise accurately grows exponentially!

To approximate a (Lipschitz) continuous function $f\colon \mathbb{R}^d \to \mathbb{R}$ with ϵ accuracy one needs $O(\epsilon^{-d})$ samples

https://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/

A cornerstone in understanding the challenges of machine learning and deep learning: the Curse of Dimensionality.

Let's start by acknowledging that the term itself sounds a bit dramatic, but it aptly conveys the complications that arise when we deal with high-dimensional data.

Our discussion today is inspired by a question on StackExchange that approaches this topic from a mathematical standpoint. So, what does it all mean?
https://stats.stackexchange.com/questions/451027/mathematical-demonstration-of-the-distance-concentration-in-high-dimensions

First, we have "Function Approximation." Imagine you have a function, let's call it f, that maps data from a d-dimensional space to a real number. Your aim is to approximate this function with a specific level of accuracy, denoted as ϵ.

This brings us to the concept of "Sample Explosion." The number of samples required to achieve this level of accuracy, ϵ, grows exponentially with the dimension, d. Mathematically, this is represented as O(ϵ^−d).

What does this mean in practical terms? Consider a 1D space; perhaps you're trying to approximate a line. If you need 10 samples for ϵ-level accuracy in 1D, you'll need 100

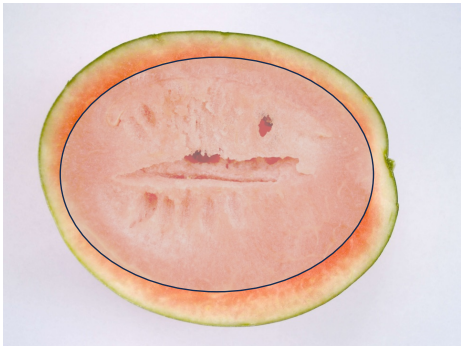
samples in 2D, and potentially 1,000 in 3D. The sample requirement explodes as dimensions increase, making the task computationally burdensome.

And why should we care, especially in the context of machine learning? This exponential growth in required samples poses significant challenges when training models on high-dimensional data. More dimensions mean more features, and without enough samples, models can either overfit or fail to recognize underlying patterns in the data.

In summary, the 'curse' in 'Curse of Dimensionality' refers to these computational and data-specific challenges that we inevitably confront as dimensionality increases. It's a constant reminder to be cautious about adding too many features without fully grasping the complexity that they introduce.

Intuition

- Let’s analyze a Pizza
- And a water melon
- Shrink by α



To explain, we're going to talk about pizzas and watermelons!
So, let's start with a pizza analogy. It's said that more than half of a pizza lies near its edge, outside the shaded area. Interestingly, the width of this 'crust' is only 18% of the diameter of the pizza. Seems a bit weird, right?
Now, consider a thick-skinned fruit like a watermelon. In such fruits, the majority of the volume is actually in the skin, not the juicy part inside.
What's the relevance? Both examples illustrate a counterintuitive geometric principle: In higher dimensions, the majority of the "mass" or "volume" of an object can reside near its boundary.
So, how can we generalize this? We can model various objects like the border of a map, the crust of a pizza, or the skin of a fruit by supposing that their basic shapes have been uniformly shrunk by some factor, let's call it α. The 'crust' or 'rind' is essentially what lies between these two similar shapes.
Why should you care? In high-dimensional spaces, like those we often encounter in deep learning, this geometric insight has far-reaching consequences. For example, when we are working with high-dimensional data, most data points are likely to be closer to the edge of the data space rather than to any other data point. This phenomenon has implications for clustering, outlier detection, and the training of neural networks.
Understanding how geometry behaves in higher dimensions helps us understand some of the counterintuitive challenges we face in deep learning and gives us a valuable lens through which to interpret our models and data.

https://stats.stackexchange.com/questions/451027/mathematical-demonstration-of-the-distance-concentration-in-high-dimensions

Intuition

- In n dimension the n-dimensional volume of the interior will be α^n times the volume of the original shape.
- The volume of the rind relative to the original volume therefore is
$$1 - \alpha^n$$
- As a function of α its rate of growth is
$$d(1 - \alpha^n) = -n\alpha^{n-1}d\alpha$$
- Beginning with no shrinking (α=1) and noting α is *decreasing* (dα is negative), we find the initial rate of growth of the rind equals n.
- This shows that the volume of the rind initially grows much faster -- n times faster -- than the rate at which the object is being shrunk.
- in higher dimensions, relatively tiny changes in distance translate to much larger changes in volume.

Let’s have a look at high-dimensional spaces and unpack the surprising behaviours they exhibit, particularly with regards to volume. This is crucial knowledge as we venture deeper into the realms of deep learning.

1. High-dimensional Volumes:
   - We begin by understanding volume in an n-dimensional space. When we scale or "shrink" an object in this space by a factor α, its volume becomes a factor of α^n times the volume of the original shape.
   - This is straightforward enough, but the implications, as we'll see, are profound.

2. Concept of the 'Rind':
   - Visualize the rind as the boundary or the "shell" that surrounds our object. To determine its volume, we subtract the volume of the shrunken object from the original, giving us (1-α^n).
   - The rind represents the difference between the original and the shrunken volumes, and it behaves rather unexpectedly in higher dimensions.

3. Rind's Rate of Growth:
   - Delving deeper, we can determine how the volume of this rind grows as we adjust the shrinking factor, α. Mathematically, this growth rate is defined by (d(1 − α^n) = -$n\alpha$^(n−1) $d\alpha$).

- Here's where it gets interesting: When we start with no shrinking (that is, α equals 1) and gradually begin to decrease α, the rind's volume grows at an initial rate of n times.

4. Key Insight:
- This essentially means the rind's volume surges n times faster than the rate at which we're shrinking the object. This behavior is a bit counter-intuitive but pivotal to understanding data in high dimensions.

5. Implications for High Dimensions:
- The takeaway? In high-dimensional spaces, even minute changes in distance can have magnified effects on volume. A tiny change in one dimension can lead to a substantially larger change in volume.
- As an example, in 2D or 3D spaces (which we're familiar with), the interior volume is (α^n) times the original volume.

6. Reiteration & Relevance to Deep Learning:
- We'll reiterate this because of its significance: In higher dimensions, minuscule changes in distance can lead to vast changes in volume. For machine learning practitioners, this has massive implications on data distribution, sampling, and model generalizability.

Intuition

- If the salami is uniformly spread out over a high dimensional pizza
  - What proportion of the salami is near the boundary?
  - i.e. how much should we shrink the pizza to e.g. make it half of its volume, say half length like half-life of radioactive elements
  - The half-length is α, solve

$$\alpha^n = \frac{1}{2};\ \alpha = 2^{-1/n} = e^{-(\log 2)/n} \approx 1 - \frac{\log 2}{n} \approx 1 - \frac{0.7}{n}$$

- 2D Pizza: half-length is 1−0.35
  - *half of the area of a pizza (n=2) lies within (approximately)* 35/2 = 18% *of its diameter from the boundary.*
- *3D Pizza:* half-length is 1−0.23
  - *half the volume lies within* 12% *of its diameter from its boundary.*
- **In very large dimensions the half-length is very close to 1**
  - *n=350 dimensions it is greater than 98%*
  - *Thus, expect half of any 350-dimensional pizza's salami to lie within 1% of its diameter from its boundary*

Deep Learning – Bernhard Kainz

Let's delve into a rather delicious problem: how salami behaves on a high-dimensional pizza. While it may sound like we're talking about dinner plans, this is actually a great way to understand data behavior in high dimensions—a concept crucial to machine learning and deep learning.

Salami on Pizza: The Setting

Imagine a pizza where the salami is uniformly spread out. We want to figure out how much of the salami is close to the boundary of the pizza in a high-dimensional space. This leads us to the concept of "half-length," a term borrowed from the idea of the half-life of radioactive elements.
Defining Half-Length (α)

To make our pizza half its original volume, we shrink it by a factor of α. The mathematical representation of this is α^n = 1/2, where n is the dimension.
Solving this equation gives α = 2^(-1/n) ≈ 1 - log 2/n, or approximately 1 - 0.7/n.
Salami in 2D and 3D

For a 2D pizza, the half-length α is 1 - 0.35. So, about 18% of the pizza's diameter will contain half the area (and salami).
For a 3D pizza, α is 1 - 0.23. Half the volume lies within just 12% of its diameter from the

boundary.
Extreme Dimensions

Now, let's think really big. In 350 dimensions, α is greater than 98%. That means, in 350 dimensions, almost all of the salami is within 1% of the pizza's diameter from its boundary.
The Cartographer's Quandary

The core question here is: "What proportion of this dataset is near the boundary?" This becomes especially relevant when we think about how data is spread in machine learning.
If your data is uniformly spread across dimensions, understanding how it behaves near the boundaries is crucial for model training and generalization.
Caveat

This generalization holds unless the data is strongly clustered. If your data points tend to stick together in certain regions, these calculations may not be accurate.
Relevance to Deep Learning

These insights have significant implications for understanding data distribution in high dimensions, which is vital for tasks like anomaly detection, clustering, and even in designing neural network architectures.

Intuition

- Without strong clustering, in higher dimensions n we can expect most Euclidean distances between observations in a dataset to be very nearly the same and to be very close to the diameter of the region in which they are enclosed. "Very close" means on the order of $1/n$.

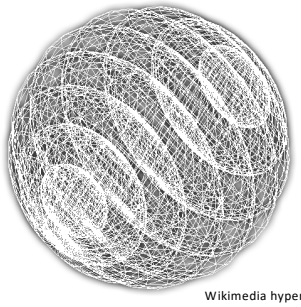
Our starting point is the idea that in higher dimensions, n, most Euclidean distances between observations will be nearly the same, and they'll be very close to the diameter of the region enclosing them.
What Does "Very Close" Mean?

When we say "very close," we are actually talking in terms of $1/n$. So, as dimensions increase, this "closeness" becomes more pronounced.
Absence of Strong Clustering:

An important qualifier here is "without strong clustering." If the data points are not strongly clustered, the above statements hold true.
Intuition Behind the Statement:

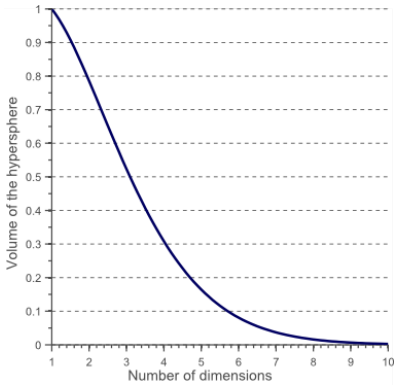
As the number of dimensions increases, the points tend to spread out in such a way that the difference in distances between various pairs of points becomes increasingly insignificant.
Imagine a cloud of points. In higher dimensions, the outer layer of that cloud is what dominates the Euclidean distance metrics.

Intuition



Wikimedia hypersphere

$$V_{sphere}(d) = \frac{\pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2}+1\right)2^d} \backsim O(c^{-d})$$

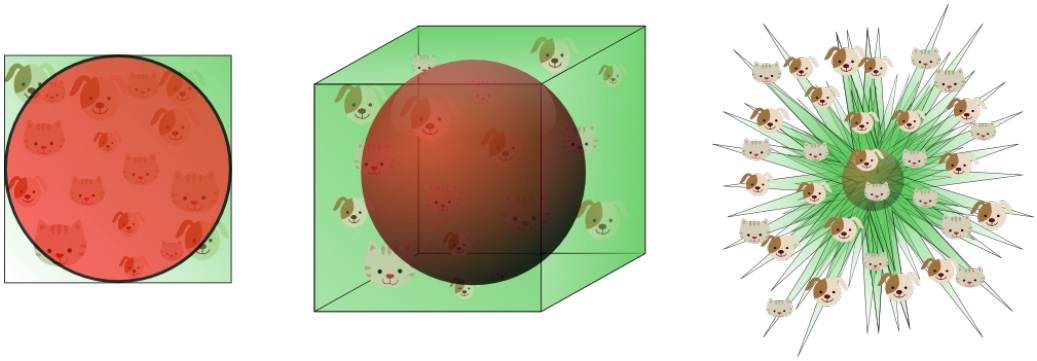


https://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/

The higher dimensional the feature space the more training samples will be in the corners of the hypercube, thus generalisation suffers.

Deep Learning – Bernhard Kainz

In the figure, you'll see a representation that illustrates how the volume of a hypersphere changes relative to the hypercube that contains it, as the number of dimensions increases.
The core takeaway is that as we go from 2D to 3D to even higher dimensions, the volume of the hypersphere starts to shrink significantly, becoming almost negligible compared to the volume of the hypercube.
This has a profound implication for machine learning: as the dimensionality of our feature space increases, most of our data points will reside in the corners of the hypercube.
This makes the task of generalizing from our training data to new, unseen data much more challenging.
The figure serves as a powerful, visual explanation of why the "curse of dimensionality" is a very real problem in high-dimensional learning tasks.
The figure also shows how the number of corners in a hypercube exponentially grows with dimensionality, accentuating the problem.
For example, an 8D hypercube has 256 corners, which can intuitively demonstrate how data points could get isolated in high-dimensional spaces, making it harder to classify or make any meaningful inferences.
So, as we look at the figure, let's keep in mind that it's not just a mathematical curiosity but a vivid depiction of a challenge that we regularly encounter in the realm of machine learning and data science.



https://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/
https://stats.stackexchange.com/questions/451027/mathematical-demonstration-of-the-distance-concentration-in-high-dimensions

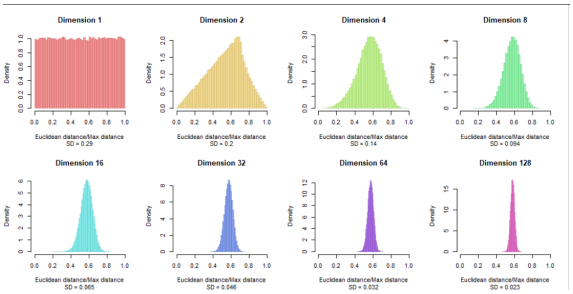
Here is another attempt to visualize this. The higher the dimensions, the more likely it gets that every sample lives in its own corner.

For example for an 8-dimensional hypercube, about 98% of the data is concentrated in its 256 corners.
As a result, when the dimensionality of the feature space goes to infinity, the ratio of the difference in minimum and maximum Euclidean distance from a sample point to the centroid, and the minimum distance itself, tends to zero.

Therefore, distance measures start losing their effectiveness to measure dissimilarity in highly dimensional spaces.
Since classifiers depend on these distance measures (e.g., Euclidean distance, Mahalanobis distance, Manhattan distance), classification is often easier in lower-dimensional spaces where fewer features are used to describe the object of interest.
Similarly, Gaussian likelihoods become flat and heavy-tailed distributions in high-dimensional spaces, such that the ratio of the difference between the minimum and maximum likelihood and the minimum likelihood itself tends to zero.
These points highlight why feature selection and dimensionality reduction techniques are often crucial in high-dimensional machine learning tasks.

Intuition

- Unit cube is asymmetric.
- To remove the asymmetry, roll the interval around into a loop where the beginning point 0 meets the end point 1: *d*-torus in *n* dimensions
- Plot distribution of normalized distance between different samples in different dimensional space
- This normalization has centered the histograms near 0.58
- around any given point on a high-dimensional torus nearly all other points on the torus are nearly the same distance away!



The unit cube is in fact asymmetric and not the greatest approximation.
To mitigate this issue, one approach is to roll the interval into a loop, where the starting point at 0 meets the end point at 1.
This creates what is known as a d-torus in n-dimensional spaces.
When we plot the distribution of normalized distances between different samples in this multi-dimensional space, an interesting pattern emerges.
The normalization process centers the histograms around a value of approximately 0.58.
Now, one key insight to draw from this is that around any given point in a high-dimensional torus, almost all other points are nearly the same distance away.
This is a counterintuitive but important observation, especially for machine learning tasks that depend on distance metrics.

Curse of dimensionality

To approximate a (Lipschitz) continuous function $f\colon \mathbb{R}^d \to \mathbb{R}$ with ϵ accuracy one needs $O(\epsilon^{-d})$ samples

Input image resolution = 12 Mpixel * 3 channels = 36M elements
With $\epsilon \sim 0.1$, we need $10^{36000000}$ samples to approximate this function space
(10^{78} to 10^{82} atoms in the known, observable universe)

Again: To approximate a Lipschitz continuous function f: $\mathbb{R}$^$d \to \mathbb{R}$ with ϵ accuracy, one needs $O(\epsilon$^(−d)) samples.
To put this into perspective, consider an input image with a resolution of 12 megapixels and 3 color channels, resulting in 36 million elements.
With an ϵ value of approximately 0.1, you would require an astounding 10^36 samples to approximate this function space accurately.
For context, that's a number ranging from 10^78 to 10^82, which is more atoms than there are in the known, observable universe.
Moving on, another intriguing point to consider is how the volume of a circle or hypersphere changes relative to the volume of the square or hypercube as we increase the dimensionality of the feature space.
Understanding these mathematical phenomena isn't just an intellectual exercise; it has practical implications for the complexity and feasibility of machine learning algorithms in high-dimensional spaces.

Invariance and Equivariance

Let’s talk about another important concept of general learning systems: Invariance and Equivariance

Invariance and equivariance

- Shift invariance

Predictor: 'cat'

This image contains a cat

Invariance and equivariance

- Shift invariance

Predictor: 'cat'

Deep Learning – Bernhard Kainz

I don't care where in the image this cat is located. I would like an image discriminator to always give as output 'cat'

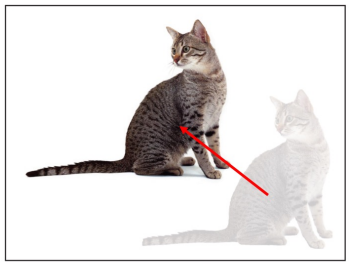
Shift invariance is a property that describes a system's unchanging response when the input is shifted.
In the context of image processing or computer vision, this means that the features of the cat in the image should be recognizable regardless of its position in the frame.
This property is critical in many deep learning applications, particularly in convolutional neural networks (CNNs), where we want the network to recognize the cat whether it's in the corner of the image or right in the center.
Understanding shift invariance helps us appreciate why certain algorithms, such as CNNs, are effective at tasks like object recognition.
So as you look at our slide showing various shifts of the cat image, remember that the goal of many machine learning models is to be as invariant to such shifts as possible.
This invariance allows models to generalize better from their training data to new, unseen data.
Shift invariance is not just a theoretical concept but a practical necessity for robust machine learning models.

Shift invariance

Input $\mathbf{x}$

Output $f(\mathbf{x}) = 1$

Shifted input $S_v\mathbf{x}$

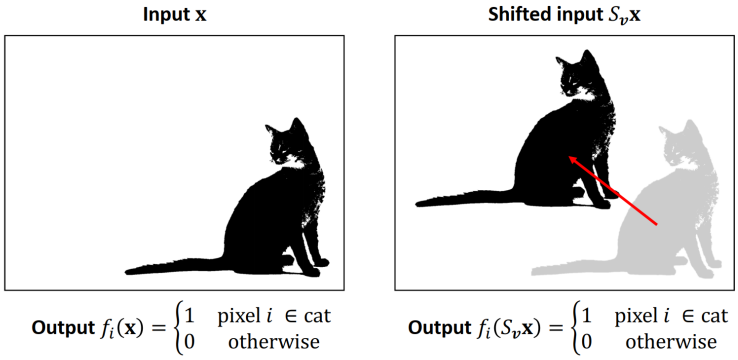


Output $f(S_v\mathbf{x}) = 1$

- **'Cat detector'** $f: \mathbb{R}^d \rightarrow \mathbb{R}$

Shift invariance means that if you shift an object in the input space, say move a cat to a different position in an image, the output from the neural network remains the same.
Let's introduce some notation to make this easier to understand.
We'll call the shift operator "S," denoted with a subscript "v," where "v" defines the vector by which we've shifted the object—in our case, the cat.
When we apply this shift operator to the input, we're essentially transforming its coordinates in the input space.
This is critical for real-world applications, where the object of interest could be anywhere in the field of view.
It's worth noting, just to clear up any misconceptions, that CNNs are not "thinking" machines.
They're complex function approximators, trained to produce a certain output for a given input.
So, when we talk about shift invariance, we're really talking about the network's ability to consistently identify features of interest, like our cat, regardless of their position in the input space.
To sum up, shift invariance is a critical property for many machine learning applications, and understanding how to achieve it can significantly improve your model's performance.

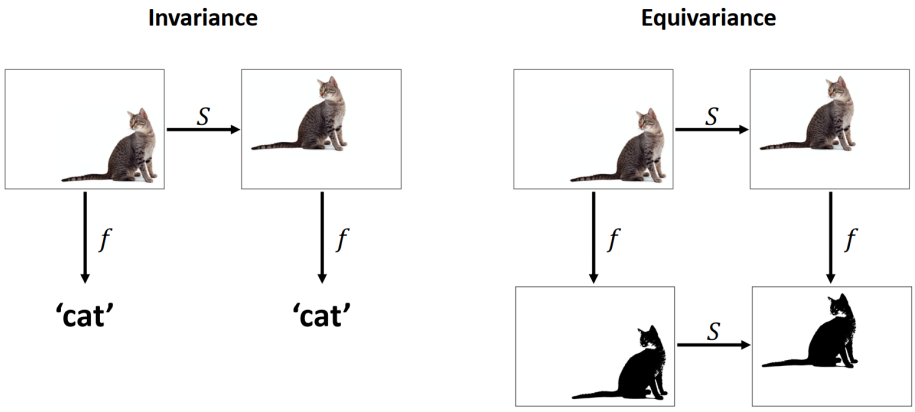
Shift equivariance



- **'Cat segmentor' f**: $\mathbb{R}^d \rightarrow \mathbb{R}^d$
- **Shift operator** S_v: $\mathbb{R}^d \rightarrow \mathbb{R}^d$ shifting the image by vector $\boldsymbol{v}$

In contrast to shift invariance, where the output remains the same regardless of how the input is shifted, shift equivariance operates a bit differently.
Imagine a model that segments an image, marking every pixel that belongs to a cat as "1" and all other pixels as "0."
In this scenario, if the cat in the image moves, the segmented output should also shift in the exact same way.
Simply put, shift equivariance means that applying the shift operator after the function yields the same result as applying the function after the shift.
To put it formally, s∘f(x) should be the same as f∘s(x).
This implies that the function f and the shift operator S commute with each other.
In essence, shift equivariance is about ensuring that your model not only recognizes but also accurately tracks the changes in position of objects within the input space.
This concept is crucial for tasks like object detection and segmentation, where the relative positions of objects in the image matter.
So, as we move forward in this course, understanding the nuances between shift invariance and shift equivariance will provide a deeper understanding of how models interact with data.

Invariance vs equivariance



Output is the same

Output is shifted like the input

Deep Learning – Bernhard Kainz

To put it simply, invariance is about stability.
When we talk about invariance, what we mean is that no matter how the input is transformed, the output should remain constant.
This is incredibly useful for tasks where the recognition of an object is more important than its position in the input space.
Now let's switch gears and talk about equivariance.
Equivariance is about consistent transformation.
In an equivariant system, the transformation applied to the input is exactly the same as the transformation applied to the output.
For example, if we move an object within an image, the corresponding output should move in the same way.
Understanding the difference between these two can significantly affect how well your model performs, depending on the task you're tackling.
So, as we delve deeper into complex models and tasks, keeping the concepts of invariance and equivariance in mind will be crucial.

Inductive bias/assumptions

- First principle: translation invariance
  - a shift in the input should simply lead to a shift in the hidden representation
- second principle: locality
  - we believe that we should not have to look very far away from any location (i,j) in order to glean relevant information to assess what this area contains

Another important tool is the concept of inductive bias or the assumptions that guide the learning process in machine learning models.
The first principle we'll discuss is translation invariance.
In the context of neural networks, this principle means that if we shift the input, the shift should propagate through to the hidden representation in a predictable manner.
In other words, the essence of what is being represented should not change simply because its location in the input space has changed.
Now, let's talk about the second principle, which is locality.
The idea behind locality is that you shouldn't have to consider far-off information to understand what's happening at a specific location in your data.
For instance, if you're looking at an image, the pixels immediately surrounding a location (i, j) should provide sufficient context to determine what that area of the image contains.
Both of these principles are deeply embedded in many of the architectures we use in deep learning, particularly in convolutional neural networks.
So as we explore more complex models, keep these principles in mind, as they fundamentally guide how these models learn from data.

translation invariance and locality – sliding window

Subimage $\hat{I}_{i,j} = I[i: i+m, j: j+n]$

- Correlation

$$C_{i,j} = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} \hat{I}_{i,j}(x,y) \cdot K(x,y)$$

Kernel (template) k

George

Image I

Deep Learning – Bernhard Kainz

How can the principles of translation invariance and locality can be practically applied using a sliding window approach.
In this approach, we define a subimage =I[i:i+m,j:j+n] that slides over the original image.
The idea is to apply a correlation operation between this subimage and some kernel K(x,y) to analyze localized regions of the image.
The correlation formula can be mathematically defined as shown on the slide.
This is a key part of many image recognition algorithms, and it embodies both the principles we've discussed so far.

Let's assume we're building a naïve algorithm to find a person in an image.
The first principle of translation invariance tells us that it doesn't matter where in the image the person is located.
Our algorithm should be able to detect the person no matter where they are in the image.

The second principle, locality, comes into play here too.
We are operating under the assumption that all the information needed to identify the person is found in a local neighborhood of pixels.
We don't need to analyze the whole image to find the person; we just need to look at localized regions.

These principles form the foundation for many complex algorithms in image recognition and are key to understanding how convolutional neural networks operate.

translation invariance and locality – sliding window

Subimage $\hat{I}_{i,j} = I[i:i+m, j:j+n]$

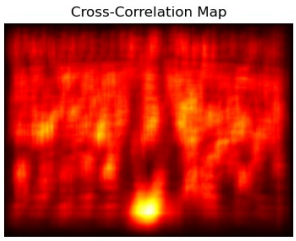
- Correlation

$$C_{i,j} = \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} \hat{I}_{i,j}(x,y) \cdot K(x,y)$$

Kernel (template) k

George

Image I

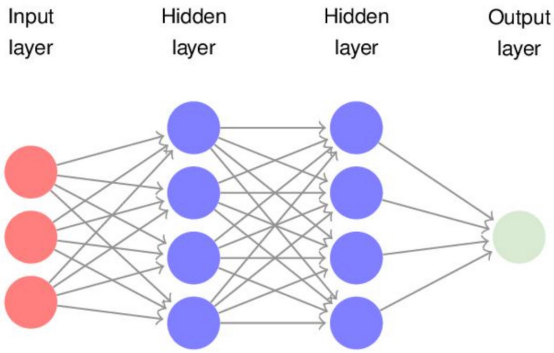


The top image now is a heat map of this function across the image and the bottom an animation about the sliding window approach and the numeric value at some locations.
Do you think that this will find George for us?
No, this approach on its own is not robust even if the kernel is directly taken from the original image.
We need high-level feature descriptors.

Convolutions

Fully connected neural networks

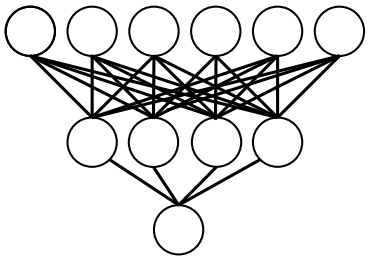
- Each input is connected to each node
- Can represent any kind of (linear) relationship between inputs

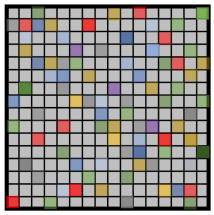


Deep Learning – Bernhard Kainz

Before we dive into CNNs, let's take a moment to recall how a fully connected neural network operates.
In a fully connected network, each input is connected to each node in the subsequent layer.
This means that every single input feature influences every single neuron in the next layer.
This architecture is extremely flexible.
It can capture any kind of relationship between inputs, particularly linear relationships.
However, this flexibility comes at a cost, including computational complexity and the potential for overfitting.
Understanding how fully connected networks function will give us a good foundation for grasping why convolutions offer a more efficient and effective alternative for certain types of data, like images.
So keep this in mind as we delve into the specifics of CNNs in the slides that follow.

Fully connected neural networks





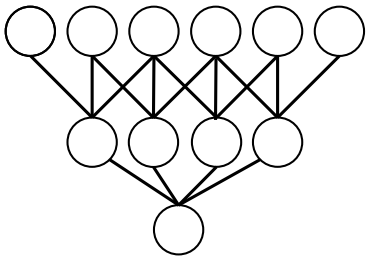
$y_j = w_{j,1}x_1 + \cdots + w_{j,n}x_n$

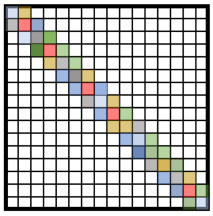
lingo:
'intractable' =
hard to control or
deal with

n^2 parameters, e.g., $36M^2$ parameters!

Now that we understand the basics of a fully connected neural network, let's talk about its limitations, especially when dealing with high-dimensional data like images.
In a fully connected layer, each node in one layer connects to each node in the next layer.
The formula here yj=wj,1x1+…+wj,nxn, helps us understand the relationship between the weights and the input features.
This setup leads to a large number of parameters, represented by n^2.
Just to give you an example, consider an input with 36 million elements, like a high-resolution image.
In this case, we're looking at 36M^2 parameters, which is an astronomical number!
This not only makes the model computationally expensive but also raises the likelihood of overfitting.
And for tasks like image recognition, where the data can be very high dimensional, this becomes a significant drawback.
Remember this problem as we discuss the advantages of using convolutions in the next slides.

Sparsely connected neural networks





Early work, e.g., Y. LeCun et al., did this

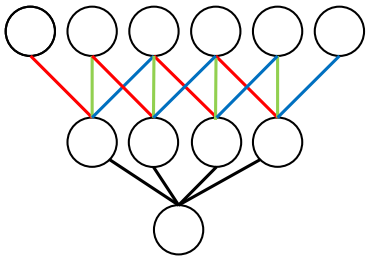
$$y_j = w_{j,i-1}x_{i-1} + w_{j,i}x_i + w_{j,i+1}x_{i+1}$$

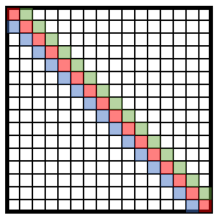
Each input neuron is connected to a small number k of hidden neurons.
Sparse connections: k*n parameters, e.g., 3*36M parameters!

Deep Learning – Bernhard Kainz

Now, let's explore how we can reduce the number of parameters in our model without losing its ability to capture important features.
One way to do that is to use sparsely connected neural networks.
In this architecture, each input neuron is connected to only a small number *k* of hidden neurons, rather than to every neuron in the hidden layer.
So what does this mean in terms of the number of parameters?
Well, now we have *k*×*n* parameters instead of *n*^2.
For instance, if *k*=3, and you have an input with 36 million elements, then you'll only need 3×36□3×36*M* parameters, which is a huge reduction.
Early work in this area, like that done by Yann LeCun and colleagues, utilized sparse connections to build more efficient networks.
It's this foundational concept that paves the way for convolutional neural networks, which we'll dive into shortly.

Weight sharing neural networks





$$y_j = w_{-1}x_{i-1} + w_0x_i + w_{+1}x_{i+1}$$

lingo:
'weight sharing'
= a subset of
weights are
identical

Each input neuron is connected to a small number k of hidden neurons and weights are shared
Shared weights (position independent): k parameters, e.g. 3 parameters!

Deep Learning – Bernhard Kainz

Now let's introduce another concept that takes this reduction to the next level: weight sharing.
In the equation yj=w−1xi−1+w0xi+w+1xi+1, the same weights w−1,w0,w+1 are reused for different input neurons. This is what we refer to as 'weight sharing.'
So, what does weight sharing mean? It means that a subset of weights are identical, not just sparse. In other words, the same set of weights is used across multiple connections.
In this model, each input neuron is connected to a small number k of hidden neurons, just like in the sparsely connected networks we discussed earlier. However, this time the weights are shared among those connections.
This results in an even more dramatic reduction of parameters. For instance, if k=3, we only have 3 parameters that are shared. That's it, just 3 parameters, no matter the size of your input!
The image you see on the slide of a sparsely connected matrix should now look different to you: it's not just sparse; it's also sharing weights across its connections. This is one of the fundamental ideas that make convolutional neural networks so powerful and efficient.

Isn't that exactly the same as we did before with correlation?

Convolution

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \quad for\ f, g : [0, \infty) \to \mathbb{R}$$

Correlation

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau \quad for\ f, g : [0, \infty) \to \mathbb{R}$$

On this slide, we see two equations that look very similar, but they represent different operations: convolution and correlation.
Both are essential operations in the realm of image processing and neural networks, but they have subtle differences.
Convolution involves flipping the kernel before performing the element-wise multiplication and sum, whereas correlation doesn't involve flipping. In other words, in convolution, we take the mirror image of the kernel across its central point before sliding it across the image. In correlation, the kernel is slid across the image as is, without any flipping.
You might be wondering why this flipping is even necessary in convolution.
Well, the flipping is a mathematical convenience and is particularly useful when we're dealing with systems defined by differential equations.
However, for neural networks, especially CNNs, the difference between convolution and correlation is often not critical.
That's because the weights of the kernel are learned from the data, whether flipped or not.
So, in practice, many deep learning libraries actually implement correlation but call it convolution.
The key takeaway is that both operations are closely related and serve the same overarching goal: local pattern detection through shared weights.

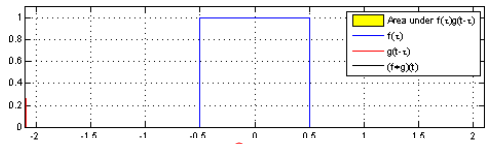
Convolution discrete version

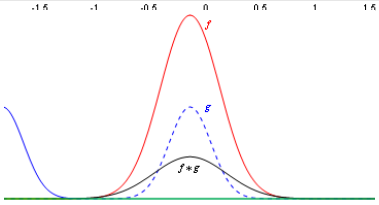
- Given array u_t and w_t, their convolution is a function s_t

$$s_t = \sum_{a=-\infty}^{+\infty} u_a w_{t-a}$$

- When either u_t and w_t are not defined, they are assumed to be 0

We are focusing on the discrete version of convolution, an essential operation in Convolutional Neural Networks.
The formula here represents how we compute a single output element, often called st, from two input arrays ut and wt.
The symbol ∑ indicates that we're summing over all possible shifts a from negative infinity to positive infinity.
Now, you might be wondering about the infinite limits in the summation.
In practice, the arrays ut and wt are finite, and for values not defined, we assume them to be zero.
So, practically speaking, the sum only includes the overlapping elements of the two arrays.
The operation involves taking the product of the elements of ut and wt at each possible shift a and summing them up to get st.
This is, in essence, a measure of similarity between ut and wt at each point t.
This concept of discrete convolution is the backbone of many of the layers in a CNN.
It allows the network to recognize local patterns in a computationally efficient way.
And the reason we focus on the discrete version is that digital signals like images are discrete entities, made up of individual pixels.





wikipedia.org

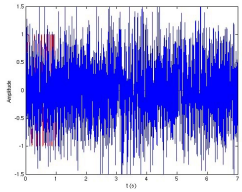
Network output (continuous):

$$(f * g)(t) = \int_0^t f(\tau)g(t-\tau)d\tau \quad for\ f, g : [0, \infty) \to \mathbb{R}$$

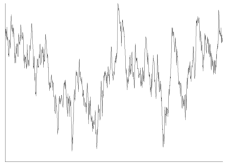
Some features of convolution are similar to cross-correlation:
for real-valued functions, of a continuous or discrete variable, it differs from cross-correlation only in that either *f*(*x*) or *g*(*x*) is reflected about the y-axis; thus it is a cross-correlation of *f*(*x*) and *g*(−*x*), or *f*(−*x*) and *g*(*x*).

Why not simply input = output for this feature detector?

Signals in the wild:



Features in the wild:



Watch:
https://www.youtube.com/watch?v=N-zd-T17uiE
https://www.youtube.com/watch?v=IaSGqQa5O-M

Deep Learning – Bernhard Kainz

We are delving deeper into the nuances between convolution and cross-correlation, two operations that look very similar but have a key difference.
Both convolution and cross-correlation involve sliding one function over another and taking the integral or sum of their product.
The key difference lies in the fact that in convolution, one of the functions is flipped before this sliding and summing operation takes place.
To put it mathematically, convolution is essentially cross-correlation, but with either f(x) or g(x) reflected about the y-axis.
So, in essence, convolution is cross-correlation between f(x) and g(−x) or f(−x) and g(x).
You may wonder why we don't simply set the input equal to the output for certain feature detectors. The answer lies in the complexity and variability of real-world signals and features.
Convolutions provide a way to adapt and generalize across different scenarios, capturing intricate details of the feature in question.
Now, let's talk about the term 'filtering,' which is often used interchangeably with convolution in signal processing. Filtering means taking an incoming signal and transforming it using a pre-defined function to get a new signal. For example, think about smoothing: we use a Gaussian kernel as the filter, and convolution operation smooths out the signal.

Some features of convolution are similar to cross-correlation: for real-valued functions, of a continuous or discrete variable, it differs from cross-correlation only in that either *f*(*x*) or *g*(*x*) is reflected about the y-axis; so it is a cross-correlation of *f*(*x*) and *g*(−*x*), or *f*(−*x*) and *g*(*x*).

Properties of convolutions

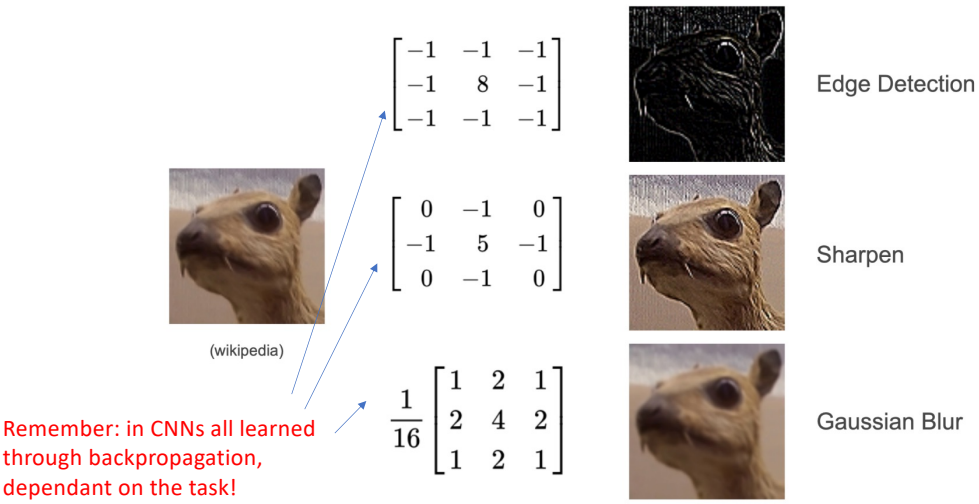
- Commutativity, $f * g = g * f$
- Associativity, $f * (g * h) = (f * g) * h$
- Distributivity, $f * (g + h) = (f * g) + (g * h)$
- Associativity with scalar multiplication, $a(f * g) = (af) * g$

Here we are focusing on some of the general mathematical properties that make convolution such a powerful and flexible operation. Understanding these properties can help us appreciate why convolution is so widely used in various applications, including Convolutional Neural Networks (CNNs).
Firstly, we have Commutativity: f∗g=g∗f. This means the order in which you convolve two functions does not matter; flipping one or the other and then sliding and summing will give you the same result.
Next, there's Associativity: f∗(g∗h)=(f∗g)∗h. This property allows us to change the grouping of functions in a series of convolutions without affecting the result. It's particularly useful in optimizing computational performance.
Distributivity comes next: f∗(g+h)=(f∗g)+(f∗h). This allows us to separate the convolution of a sum of functions into individual convolutions and then summing those results. This property can also aid in computational efficiency.
Finally, Associativity with Scalar Multiplication: a(f∗g)=(af)∗g. This means that scaling one function by a constant before the convolution is the same as scaling the result of the convolution by that constant.
These properties make convolution an algebraically rich and computationally efficient operation, qualities that are extremely beneficial when designing algorithms and architectures for deep learning, especially CNNs.

Why Convolutions for pattern-matching?

- *Historical Reasons:* The operation in CNNs resembles the discrete 2D convolution operation, even though it's technically cross-correlation. The term "convolution" in CNNs has stuck due to historical reasons and convention. Computational advantages for large kernels with FFT. Mathematical advantages for probability distributions.
- *Flipped Kernels:* In some contexts, before applying the convolution operation, the kernel is flipped both horizontally and vertically. Once flipped, applying cross-correlation will be equivalent to applying convolution with the original kernel. However, **in CNNs, the kernels are learned, so it doesn't matter if they are flipped or not**; **the network will learn the appropriate values during training**.
- Regardless of whether true convolution (with kernel flipping) or cross-correlation is used, the result of training will be the same. The network will adjust its weights based on the feedback from the loss during backpropagation. Thus, for the purpose of training neural networks, the distinction between the two becomes largely irrelevant.
- *Implementation*: In deep learning frameworks like TensorFlow or PyTorch, the operation performed in the convolutional layers is actually cross-correlation. However, they still use the term "convolution" due to convention.

Examples of 2D image filters



Slide credit: Smola, Li 2019

Here are some practical applications of convolution by discussing some classic 2D image filters. These are key building blocks in image processing and, by extension, in Convolutional Neural Networks.

First on the list is Edge Detection.

This is a filter designed to highlight the boundaries within an image. It helps the network pay attention to shapes and borders, which are often critical in tasks like object detection and segmentation.

Next, we have the Sharpen filter. This filter enhances the details in an image, making it easier for the model to recognize subtle features.

It accentuates the intensity changes, making edges and textures more pronounced.

Third is Gaussian Blur. This filter is used to smooth an image by reducing its high-frequency components.

This is particularly useful in noise reduction and can sometimes be used as a preprocessing step.

It's crucial to remember that in the context of CNNs, these filters are not manually engineered but rather learned through backpropagation.

The network adapts these filters automatically to be optimal for the specific task at hand, whether it's image classification, segmentation, or something else.

So, in essence, these classic filters serve as a starting point for understanding what kind of features CNNs can learn to recognize, but the actual filters in a trained CNN are data-driven and task-specific.

CNN building blocks

- Convolutional Layer
- Pooling Layer
- Fully Connected Layer
- Flatten Layer
- Dropout
- Batch Normalization
- Activation Function
- Loss Function
- Optimizer

Convolutional Layers.
This is the heart of a CNN, where filters or kernels slide across the input to produce feature maps. It captures local patterns and details like edges and textures, making it especially suited for image-related tasks.

Activation Functions.
Non-linearity is introduced here, allowing the model to learn complex mappings from inputs to outputs. ReLU, or Rectified Linear Unit, is commonly used because of its effectiveness in CNNs.

Pooling Layers.
This layer reduces the spatial dimensions of the feature maps, both simplifying the model and lessening the computational burden. Max-pooling is often used as it retains the most salient features.

Fully Connected Layers.
Usually situated towards the end, this layer is where each neuron connects to every neuron in the previous layer, integrating the learned features for final predictions.

Flatten Layers.
This serves as a bridge between the convolutional layers and the fully connected layers,

essentially converting 2D feature maps into 1D vectors.

Dropout can be used
This is a regularization technique that randomly deactivates a subset of neurons during training. This helps prevent overfitting and makes the network more robust.

Batch Normalization.
This keeps the distribution of each layer's outputs stable, aiding in faster and more reliable training convergence.

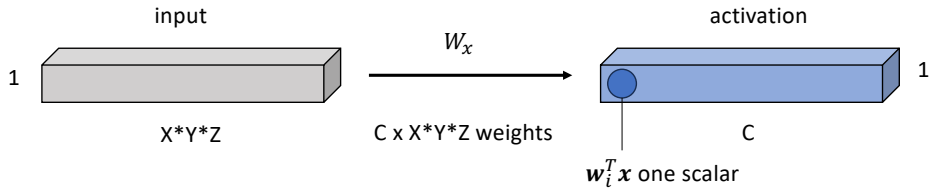
The Softmax Layer is commonly found at the end.
It converts the final layer's outputs into a probability distribution, useful for classification tasks.

For measuring performance during training and informing the backpropagation gradient, we have the Loss Function.
Cross-entropy loss is commonly used for classification tasks in CNNs. It quantifies how well the model's predictions align with the true labels.

Last but not least is the Optimizer.
This adjusts the network weights based on the gradients computed during backpropagation. SGD (Stochastic Gradient Descent) and Adam are popular choices.

Input Tensor conventional NNs

Instead of stacking a [X,Y,Z] RGB image into a X*Y*Z x 1 vector for a conventional NN categorising in C classes



- we have priors about the data!

First principle: translation invariance

a shift in the input should simply lead to a shift in the hidden representation

second principle: locality

we believe that we should not have to look very far away from any location (i,j) in order to glean relevant information to assess what this area contains

Deep Learning – Bernhard Kainz

The input tensor setup for conventional Neural Networks compared to Convolutional Neural Networks.

This sets the stage for understanding why CNNs have advantages for image-based tasks.

First, let's talk about the conventional approach.

In a typical Neural Network, you'd take an image with dimensions [X, Y, Z] for RGB channels and simply flatten it into a long vector of size XYZ x 1. This would then feed into the network for classification into C classes.

However, we have certain priors about image data that make this approach sub-optimal.

Let's dive into these priors starting with the first principle: translation invariance.

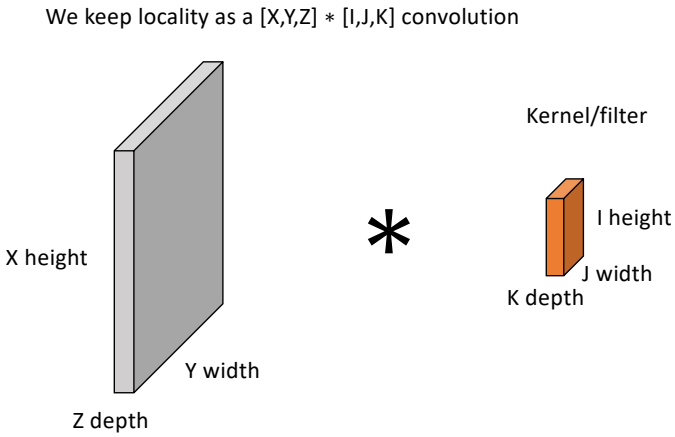
What we mean by this is that if you shift an object in an image, the intrinsic characteristics of the object don't change.

Therefore, the features learned by the network should also shift accordingly.

The second principle is locality. In image data, the relevant information for making a decision usually lies close to a particular location. It implies that we shouldn't have to scan the whole image to decide what a small region contains. In a fully connected network, this locality is not naturally accounted for, but in CNNs, it is.

By keeping these principles in mind, Convolutional Neural Networks offer a more efficient and intuitively-aligned way to process image data compared to flattening the input as we would in conventional Neural Networks.

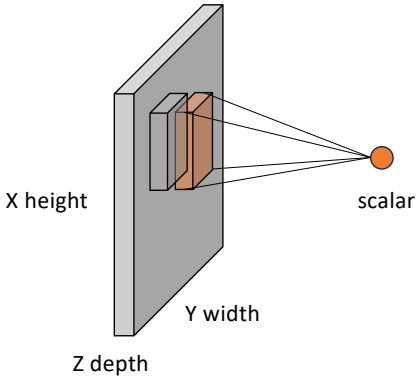
Kernel



In this slide, we're focusing on how Convolutional Neural Networks, or CNNs, preserve the principle of locality in the input tensor.
You'll notice we have a graphical representation showing an input layer with dimensions [X, Y, Z] and a filter kernel of dimensions [I, J, K]. Between them, you see a convolution symbol.
So what does this mean?
Instead of flattening the input image into a 1D vector as in a conventional Neural Network, CNNs maintain the original structure of the image as a 3D tensor of dimensions [X, Y, Z]. This helps us keep spatial relationships between pixels intact.
This 3D tensor is then convolved with a smaller 3D kernel of dimensions [I, J, K]. The idea is that this small kernel slides over the 3D input tensor to generate feature maps, preserving the local spatial relationships in the image.
This approach is in line with the principle of locality, which emphasizes that we should be able to assess what a region in an image contains by just looking at the surrounding pixels.
By using this method, CNNs inherently take into account the spatial structure of the image, making them especially powerful for image-related tasks.

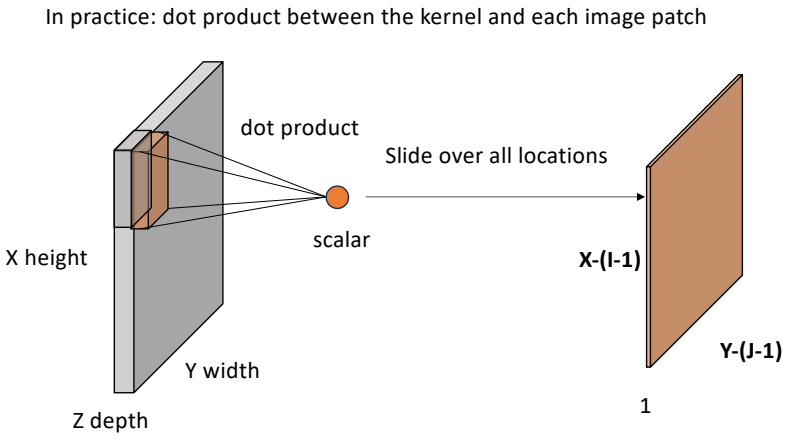
Convolution

In practice: dot product between the kernel and each image patch



In practice, convolution is often implemented as a series of dot products. We take a patch of the input image, which matches the size of our kernel, and we perform a dot product between this patch and the kernel.
This dot product is essentially a weighted sum of the pixel values in the image patch, where the weights are determined by the kernel.
The output of each of these dot products forms a single pixel in the resulting feature map.
By doing this, we capture the local spatial features from each region of the input image, and compile them into a feature map.
This method is highly efficient and is one of the reasons why CNNs are so effective for image analysis tasks. It adheres to the principles of translation invariance and locality, capturing spatial features while significantly reducing the number of parameters compared to fully connected networks.

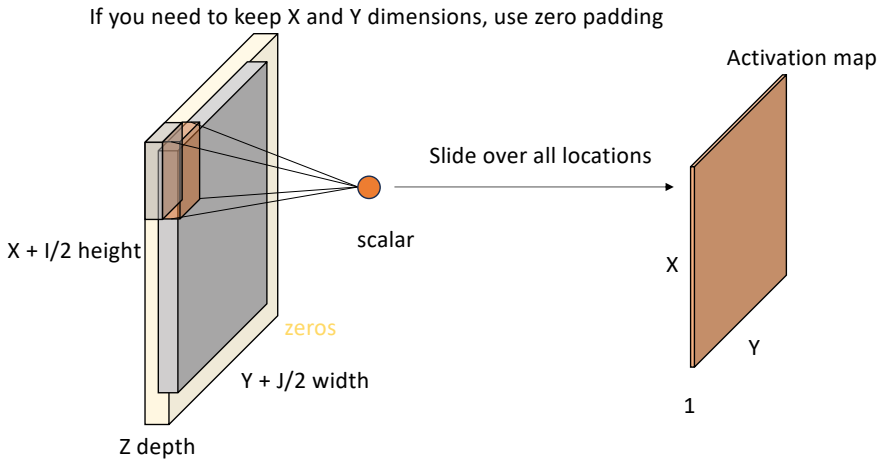
Convolution



Deep Learning – Bernhard Kainz

You'll notice an image of an input layer, a filter kernel, and the resulting output after the convolution operation. The output dimensions are reduced, specifically by half the size of the filter in both width and height.
Let's break this down.
So why are the dimensions of the output reduced?
Well, as we slide the kernel across the input image and perform dot products, we're essentially aggregating information from the original pixels into new, condensed values.
The kernel size determines how many pixels are aggregated into each new value, which also determines the size of the output.
In our example, the output dimensions are reduced by half the size of the filter. This is a common feature of convolutions, especially when we don't use padding or stride greater than one.
This reduction in dimensionality is not always a bad thing. It can actually be quite useful for reducing the computational load for future operations and can also help in abstracting the higher-level features in the image.
Remember, the primary idea is to slide this filter over all the locations in the input image to produce a feature map that captures the spatial hierarchies present in our input.
So, with each convolution operation, we're essentially simplifying the image into a form that retains essential features but is easier and faster for the neural network to analyze.

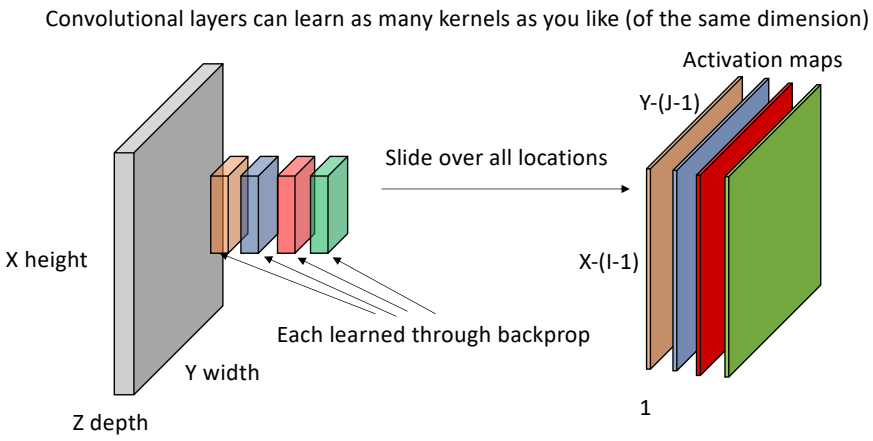
Input Tensor



Deep Learning – Bernhard Kainz

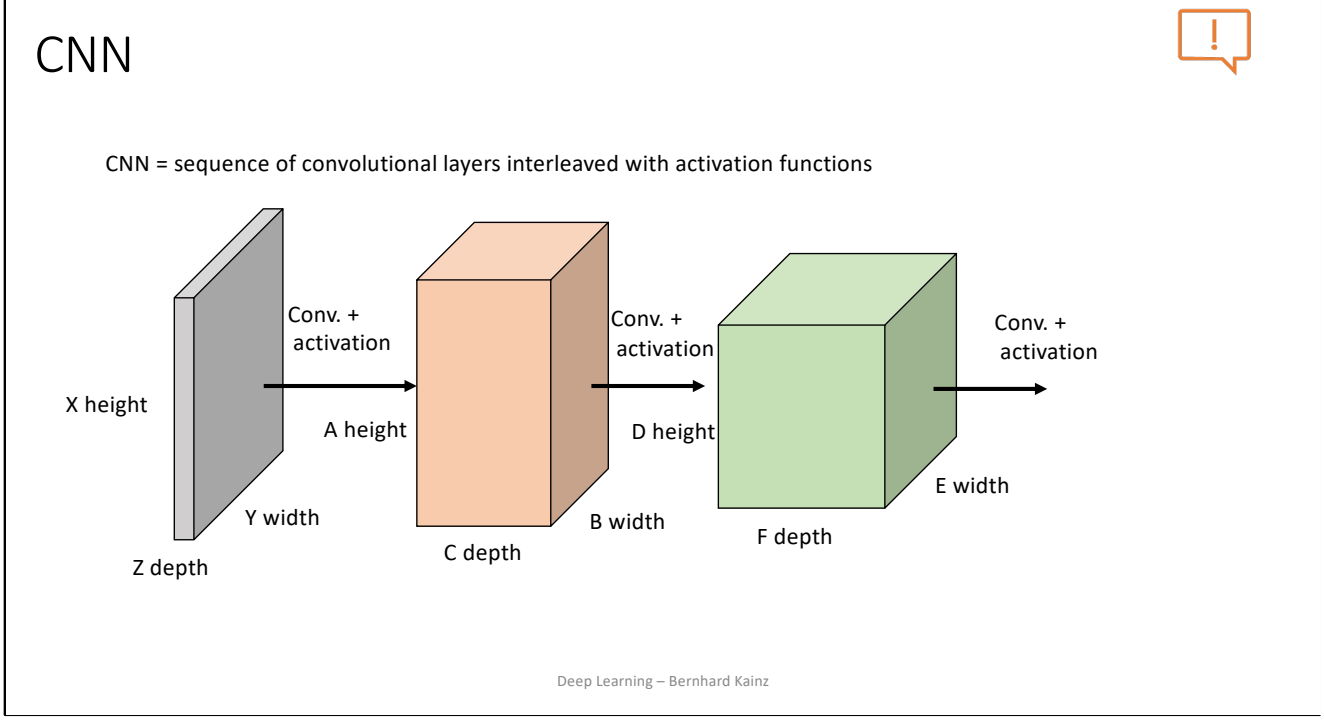
How to preserve the original dimensions of the input when using convolution operations?
You'll see here that the dimensions of the output image are the same as the original input image. How is that possible?
The answer is zero padding.
Zero padding is the practice of adding zeros around the edge of the input image before applying the convolution operation.
So why would you want to do this?
Preserving the dimensions of the input image can be beneficial for several reasons. For example, it allows for more layers to be stacked without shrinking the spatial dimensions too much, which can be particularly useful for deeper networks.
By adding a border of zeros around the original input, you essentially create a buffer that allows the filter to slide across every position it would normally pass over if it could extend beyond the input's actual borders.
This way, the output retains the same width and height dimensions as the input, ensuring that spatial relationships between features are maintained throughout the layers of the network.

Feature extraction



One key point to understand is that convolutional layers can learn multiple filters, or kernels, at the same time, and all of them are of the same dimension.
Why is this important?
Each kernel captures a different feature or pattern in the input data. For example, one might specialize in detecting horizontal edges, while another might focus on capturing color gradients.
Because these kernels are learned through backpropagation, the network automatically adjusts them during training to capture the most important features for a given task.
The images you see on the slide illustrate this concept. Each filter kernel has its own corresponding activation map, showing where in the image the particular feature it captures is located.
So, when you have multiple kernels in a single layer, you are essentially building a rich, complex understanding of the input data from various perspectives.
In essence, more kernels mean a richer feature representation. You can think of each kernel as a specialized mini-detector working in tandem with others to create a comprehensive understanding of the input.

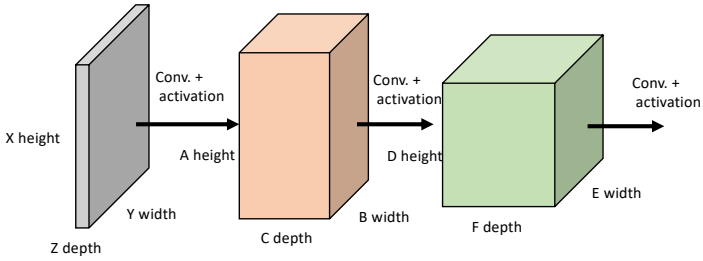
CNN



At its core, a CNN is essentially a sequence of convolutional layers, each followed by an activation function.
Why is this sequence important?
Convolutional layers are responsible for feature extraction. They take in the raw data and produce a set of feature maps that highlight various aspects of the data.
Immediately following each convolutional layer is an activation function, often a ReLU (Rectified Linear Unit).
The role of the activation function is to introduce non-linearity into the network.
Without non-linearity, the network would not be able to capture complex relationships and patterns in the data and everything would become equivalent to one linear transformation.
As we move through the layers in a CNN, the dimensions of the intermediate tensors often change.
For example, you may start with an input image that has dimensions [height, width, color_channels], and as it progresses through the layers, the dimensions may reduce or expand depending on the operations being performed.
Remember, each layer in the network is learning to recognize increasingly complex features. Early layers might capture simple patterns like edges and corners, while deeper layers might recognize more abstract features like the shape of a nose or the contour of a car.

Parameters

CNN = sequence of convolutional layers interleaved with activation functions



Each filter: I*J*K + 1 (bias) parameters to learn

Each filter in a convolutional layer is defined by its dimensions, which for our purposes are I, J, and K.

So, for a single filter, the total number of parameters is I multiplied by J multiplied by K.

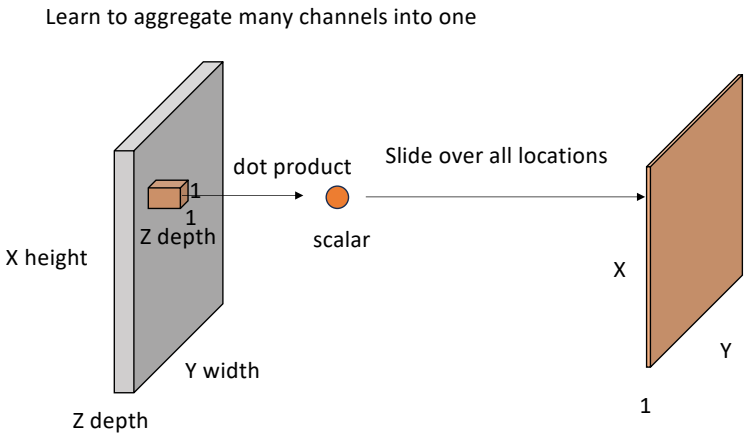
But wait, we're not done. There's one more parameter: the bias term.

Adding the bias term, we have I×J×K+1 parameters for each filter in a convolutional layer. The bias term is crucial because it allows the filter to have some flexibility, effectively shifting the activation function to better fit the data.

To give you a sense of scale, imagine you have a simple 3x3x3 filter; that's 27 weight parameters plus one bias, totalling 28 parameters for that filter alone.

When you scale this to multiple filters and layers, the number of parameters can grow quite large, but remember, this is often still much fewer than what you'd find in a fully connected network for the same task.

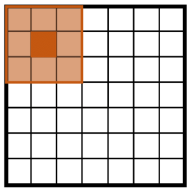
1x1 Convolution – reduce depth/NN across depth



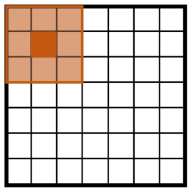
Deep Learning – Bernhard Kainz

A 1x1 convolution? At first glance, it may seem almost trivial: a single value is multiplied by a single weight and a bias is added. But its applications are far from trivial.
The main utility of a 1x1 convolution is to reduce the depth of our network. Think of it as a way to perform dimensionality reduction across the depth of the feature map.
When we have a high number of channels, or "depth", in our input volume, a 1x1 convolution can effectively condense that information.
It's essentially a dot product operation across the depth of the input volume.
As with regular convolutions, these 1x1 convolutions slide over all locations in the input.
So, in a nutshell, 1x1 convolutions allow the network to learn how to aggregate many channels into fewer channels, thereby reducing computational complexity while maintaining important features.
It's a neat trick that has a big impact on the efficiency and performance of convolutional neural networks.

Padding and strides



7x7 input
3x3 filter
stride 1
no padding



7x7 input
3x3 filter
stride 2
no padding

Figure: adapted from Fei Fei et al.

padding and strides.

Let's first talk about strides. Stride refers to the step size that the filter takes as it slides across the input image.

A stride of 1 means the filter moves one pixel at a time, which is the most straightforward case. As you can see in the example image, with a 7x7 input and a 3x3 filter, a stride of 1 gives us a 5x5 output.

However, a larger stride like 2 would mean the filter jumps two pixels at a time. In our example, this reduces the output size to 3x3.

Strides are mainly used to reduce the spatial dimensions of the output volume, which in turn reduces the amount of parameters and computation in the network.

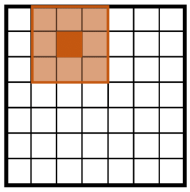
Now, let's talk about padding. Padding is the technique of adding extra pixels around the input image. In this example, we're using "no padding," meaning we're not adding any pixels around our 7x7 input.

The primary purpose of padding is to control the spatial size of the output volumes, mostly to preserve the spatial dimensions of the input volume so that they match.

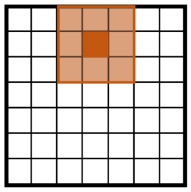
These two parameters—stride and padding—give us fine-grained control over the architecture of our CNNs and allow us to manage the computational load while still capturing the features effectively.

Understanding how to set these parameters appropriately is crucial for designing effective and efficient CNNs.

Padding and strides



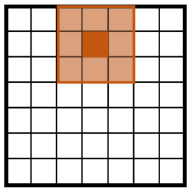
7x7 input
3x3 filter
stride 1
no padding



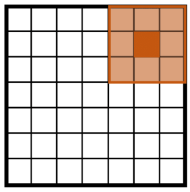
7x7 input
3x3 filter
stride 2
no padding

Figure: adapted from Fei Fei et al.

Padding and strides



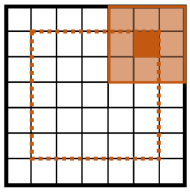
7x7 input
3x3 filter
stride 1
no padding



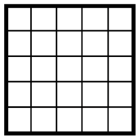
7x7 input
3x3 filter
stride 2
no padding

Figure: adapted from Fei Fei et al.

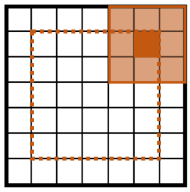
Padding and strides



7x7 input
3x3 filter
stride 1
no padding



5x5 output

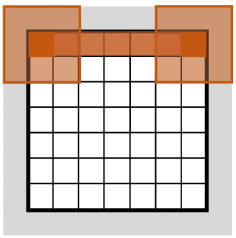


7x7 input
3x3 filter
stride 2
no padding

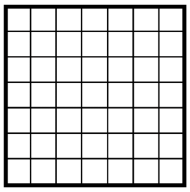
3x3 output

Figure: adapted from Fei Fei et al.

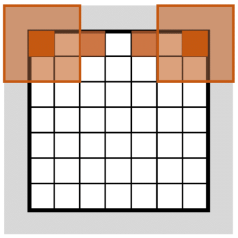
Padding and strides



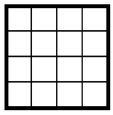
7x7 input
3x3 filter
stride 1
zero padding



7x7 output



7x7 input
3x3 filter
stride 2
zero padding



4x4 output

Figure: adapted from Fei Fei et al.

Computational complexity

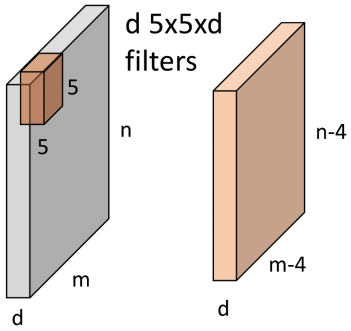


Figure: adapted from Fei Fei et al.

A 5x5 convolution with a single filter would involve 25 multiply-and-add operations per input element. If we instead use two consecutive 3x3 convolutions, the first 3x3 convolution requires 9 operations per input element, and the second 3x3 convolution also requires 9 operations on the output of the first.

Here's where it gets interesting. Even though 9 plus 9 is 18, which is less than 25, the real computational savings can be even greater in practice. That's because the first 3x3 convolution generates a smaller output size compared to the original input, and the second 3x3 convolution works on this reduced-size output.

This approach offers computational efficiency and also introduces an additional non-linearity by having two activation functions instead of one, which can help the model capture more complex features.

Factorizing a 5x5 into two 3x3 convolutions is just one example of how you can optimize the architecture of CNNs without compromising their ability to learn rich representations.

Factorized convolution

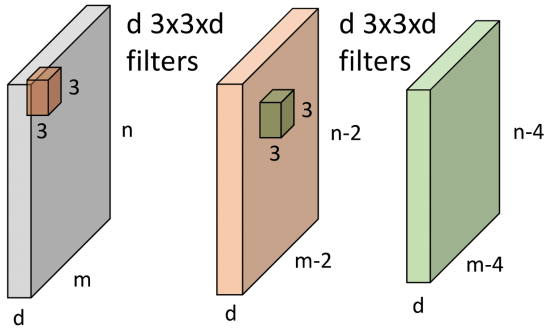


Figure: adapted from Fei Fei et al.

While two 3x3 convolutions might seem to serve as a good approximation for one 5x5 convolution, it's crucial to understand that this is indeed an approximation. You are essentially reducing the number of parameters and thus potentially the representational capacity of that layer.

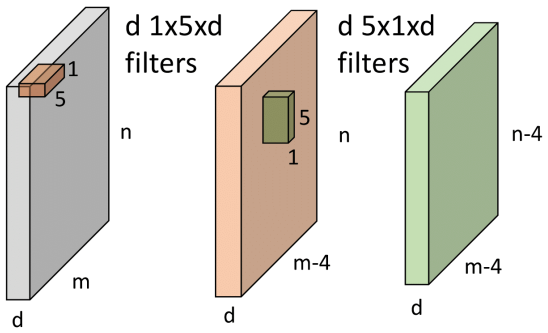
In a 5x5 convolution, you would typically have 25 parameters for each filter, excluding the bias term. When you break it down to two sequential 3x3 convolutions, each with 9 parameters, you end up with a total of 18 parameters, again excluding bias terms.

The interesting trade-off here is between computational efficiency and expressiveness. Two 3x3 convolutions are computationally less expensive but may not capture the same level of detail as a single 5x5 convolution.

Also, inserting an activation function between the two 3x3 convolutions introduces an extra non-linearity, making the approximation more capable of capturing complex features. However, this still doesn't match the 'possibilities' or parameter space offered by a single 5x5 filter.

So when you opt for this factorization, remember that you're making a trade-off: gaining computational efficiency at the potential cost of some representational power.

Separable convolution



e.g. m=n=32, d=3
32x28x3x5x3+1 +
28x28x3x5x3+1 =
75602 ops

vs. 5x5
28x28x3x5x5x3+1=
176401 ops

Figure: adapted from Fei Fei et al.

In traditional convolution, if you have a e.g. a 5x5 filter, you would slide this filter across your image to produce a feature map.
Separable convolutions aim to reduce the computational burden of this operation. Instead of a 5x5 convolution, you can approximate it by first applying a 1x5 convolution and then a 5x1 convolution. This breaks down the original 5x5 convolution into two separate operations, hence the name 'separable.'
Why is this useful? It's about computational efficiency. A single 5x5 convolution has 25 learnable parameters. But if you break it into a 1x5 followed by a 5x1 convolution, you have 5 parameters for the first convolution and 5 for the second, totalling 10 parameters. This effectively reduces the computational cost and also the number of learnable parameters, which can be very beneficial in large-scale or resource-constrained applications.
However, it's crucial to note that this is an approximation technique. The catch is that it works well only if the original 5x5 filter can be accurately approximated by the two smaller filters. In some cases, this approximation might lead to a loss of information or representational power, but in practice, the efficiency gains often outweigh the drawbacks.
So, separable convolutions offer a way to make your neural network more efficient, at the potential cost of some representational power. It's a trade-off that often makes sense when computational resources are a concern.

Pooling

- Permutation-invariant aggregation+downsampling (typically max or avg)
- Reduces resolution
- Hierarchical features
- Contributes to approximate shift/deformation invariance

6	1	2	4
1	6	7	8
3	5	2	0
1	2	3	4

Figure: adapted from Fei Fei et al.

Let's now delve into the concept of Pooling, another crucial building block in Convolutional Neural Networks. At its core, pooling serves two main purposes: it performs a kind of aggregation and it also downsamples the image or feature map.
So what do we mean by 'Permutation-invariant aggregation'? Essentially, the order in which the pixels appear in the window doesn't affect the outcome of the pooling operation. This is what contributes to its property of local translation invariance or shift invariance. Regardless of minor shifts or deformations in the input image, the pooling operation remains largely unaffected.
The most commonly used method for pooling is max pooling. In max pooling, we examine a block or grid of pixels and select the maximum value among them. This maximum value then represents that entire block in the downscaled version of the image.
Here are the key takeaways:
1.Reduces Resolution: Pooling makes the computational load lighter by reducing the spatial dimensions.
2.Hierarchical Features: As we progress through layers, pooling helps the network to concentrate on increasingly abstract features, adding a level of hierarchy.
3.Shift/Deformation Invariance: Pooling contributes to the network's robustness against small shifts or deformations in the input.
So, pooling is not just about making the network faster or lighter; it's a strategic component that adds robustness and translational invariance to the model.

Pooling

- Permutation-invariant aggregation+downsampling (typically max or avg)
- Reduces resolution
- Hierarchical features
- Contributes to approximate shift/deformation invariance

6	1	2	4
1	6	7	8
3	5	2	0
1	2	3	4

6	

Figure: adapted from Fei Fei et al.

Pooling

- Permutation-invariant aggregation+downsampling (typically max or avg)
- Reduces resolution
- Hierarchical features
- Contributes to approximate shift/deformation invariance

6	1	2	4
1	6	7	8
3	5	2	0
1	2	3	4

6	8

Figure: adapted from Fei Fei et al.

Pooling

- Permutation-invariant aggregation+downsampling (typically max or avg)
- Reduces resolution
- Hierarchical features
- Contributes to approximate shift/deformation invariance

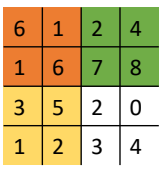


Figure: adapted from Fei Fei et al.

Pooling

- Permutation-invariant aggregation+downsampling (typically max or avg)
- Reduces resolution
- Hierarchical features
- Contributes to approximate shift/deformation invariance

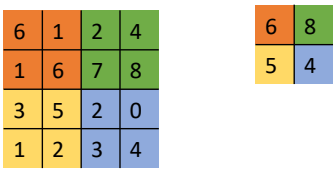


Figure: adapted from Fei Fei et al.

Pooling

- Applied to each channel separately

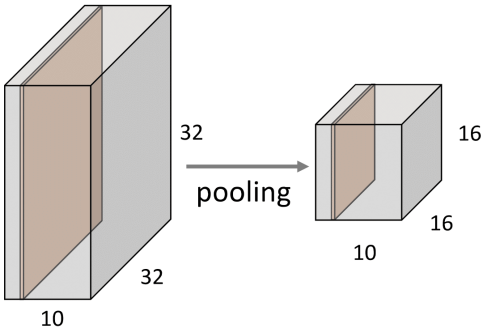


Figure: adapted from Fei Fei et al.

One key detail is that pooling is applied to each channel separately. That means if you have an RGB image with three channels—Red, Green, and Blue -- the pooling operation will be performed on each of these channels individually, not on the combined information. This helps in preserving the integrity of each feature map as we move deeper into the network.

Now, pooling can be done using different functions:

1.Max Pooling: The most commonly used form where the maximum pixel value from the grid is selected.

2.Average Pooling: Instead of picking the maximum, here we calculate the average value of all pixels within the grid.

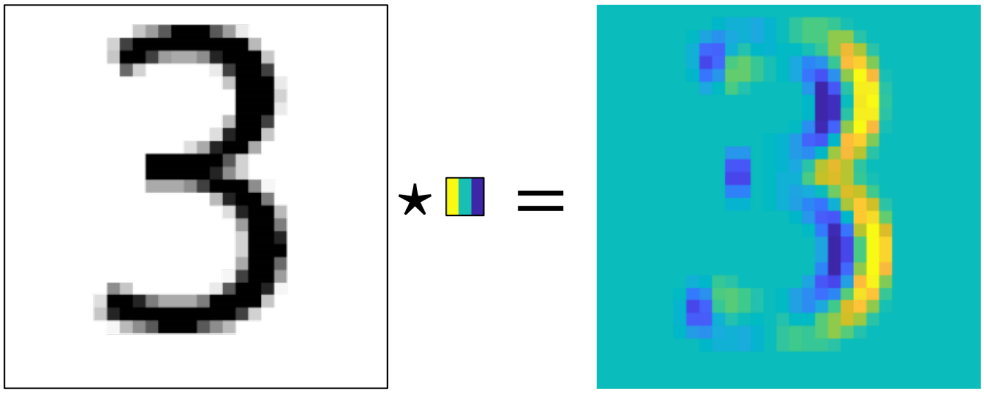
Each of these methods has its own set of advantages and use-cases. Max pooling tends to focus on the most prominent features, while average pooling provides a smoother, more generalized output.

An interesting side note is that pooling can actually be replaced with a convolutional layer with strides in some benchmark applications. Essentially, a strided convolution would perform a similar downsampling operation as pooling, but with the added benefit of learning the downsampling function. This shows the flexibility and adaptability of CNN architectures. However, using striding instead of pooling is very application-specific and may not be a universal replacement.

While pooling is a prevalent approach to downsampling and achieving translation invariance, it's not the only game in town. Depending on the specific needs of your

application, you might opt for different pooling functions or even replace pooling with strided convolutions.

Equivariance in CNNs



Output of convolutional layer (shift equivariant)

Deep Learning – Bernhard Kainz

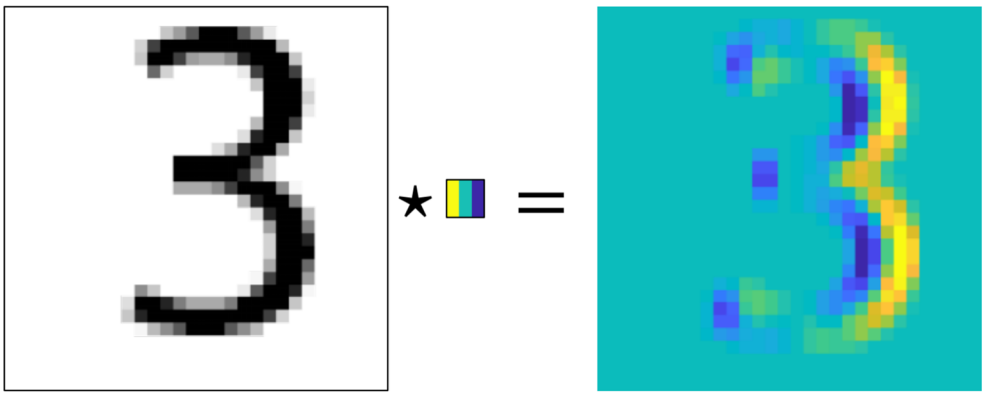
Let's dig a bit deeper into the theoretical underpinnings of convolutions, specifically regarding their property of shift-equivariance. Shift-equivariance implies that if we shift an input, the output shifts by the same amount. In other words, applying a filter kernel to an image should not fundamentally change the filter's response to the features it's designed to capture.

To understand this better, we can take a trip into frequency space via the Fourier Transform. The Fourier Transform allows us to represent our image in terms of its constituent frequencies rather than pixel values. Now, an interesting property of the Fourier Transform is that a convolution in the image domain (or time domain, if we were talking about signals) is equivalent to multiplication in the frequency domain. This equivalence establishes the foundational proof for the shift-equivariance property of convolutions.

You might remember the animations we had on convolving two functions; those were essentially depicting this principle. In essence, the convolution operation is designed in such a way that it is insensitive to where exactly in the image the feature appears—making it shift-equivariant.

To tie this back to our earlier discussions, shift-equivariance is one of the essential properties that make CNNs so effective for image recognition tasks, where the position of the feature within the image could vary.

Equivariance in CNNs

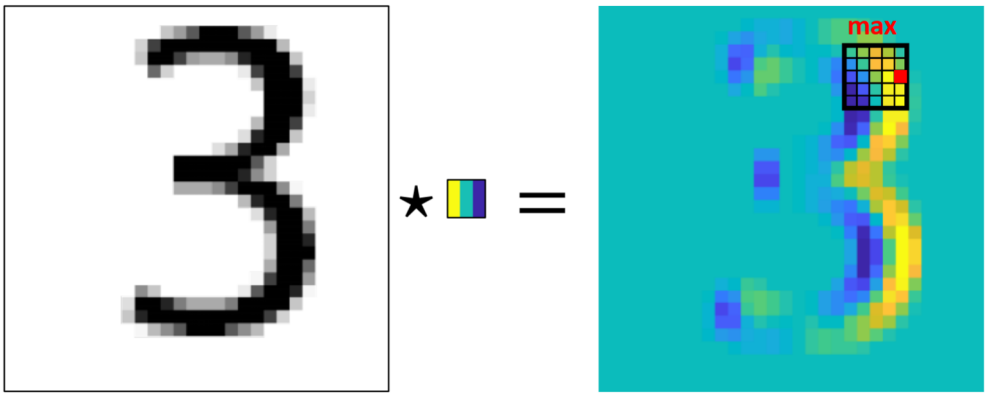


Output of convolutional layer (shift equivariant)

Deep Learning – Bernhard Kainz

So, when you have an object in an image and you move it around—perhaps you shift it a bit to the left or a little upwards—the filters in your CNN should ideally produce the same response at the new location of the object in the output feature map.
Why is this important? Well, think about real-world applications like object detection or image classification. Objects you're trying to identify could be anywhere in the frame. They might not be perfectly centered; they could be at the corner or even partially outside the frame. The shift-equivariance property ensures that the model is robust to such positional changes, making it more accurate and reliable.
The shift-equivariance property ensures that CNNs remain 'attentive' to the features they're designed to capture, regardless of where those features are positioned in the input. This is a key advantage when we're working with varied and unpredictable real-world data."

Approximate invariance in CNNs with pooling

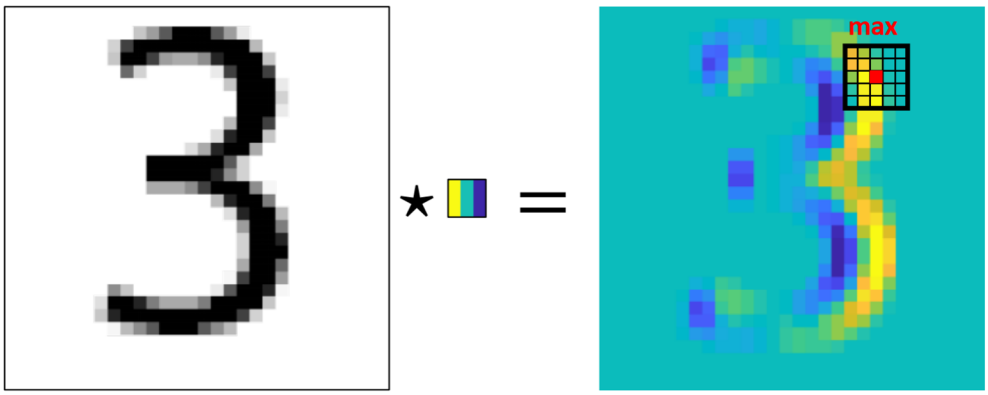


Output of convolutional layer+max pooling (~shift invariant)

Deep Learning – Bernhard Kainz

Now, let's take a look at how pooling layers contribute to what we call 'approximate shift invariance.'
In the context of CNNs, shift invariance essentially means that small translations or movements in the input image won't affect the key features that the network identifies.
Pooling, especially max-pooling, plays a big role here. Let's say you have an object in an image and that object gets shifted a little bit. When you apply max-pooling, you're looking for the maximum value in a local window of your feature map. Even if the object has shifted slightly within that window, the maximum value -- the most 'important' feature in that region -- should, in theory, remain the same.
Now, it's worth noting that this isn't a perfect system. It's what we call 'approximate' shift invariance. That's because pooling operates on a local scale. If the object shifts too far -- outside of the local window -- the maximum value could change, and that would affect the output.
But for small, local shifts, pooling provides a level of robustness. It ensures that the key features we want the CNN to recognize are not 'lost' due to minor changes in position.
So, pooling not only reduces the dimensionality of our data, making the network easier to compute, but also adds a layer of robustness against small shifts in the input.

Approximate invariance in CNNs with pooling

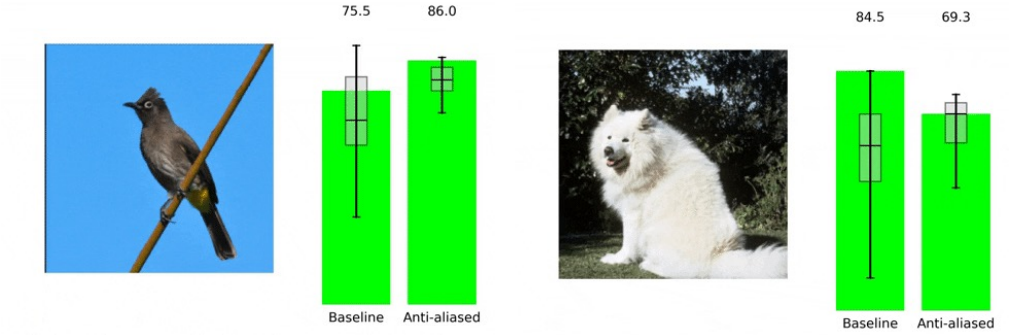


Output of convolutional layer+max pooling (~shift invariant)

Not the full story…

Nyquist sampling theorem = sample at least twice as fast to keep all information

- But striding ignores the Nyquist sampling theorem and aliases

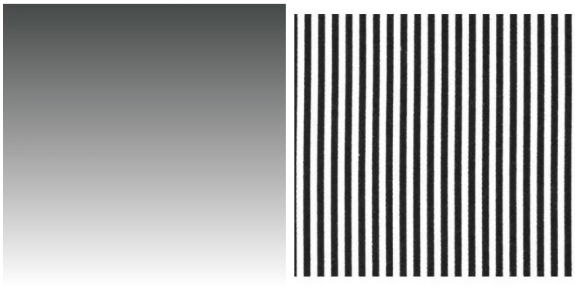


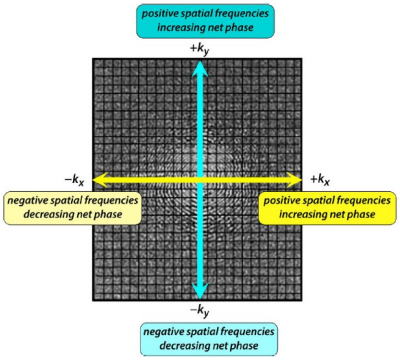
R. Zhang.
Making Convolutional Networks Shift-Invariant Again.
In ICML, 2019.

However, it's crucial to note that traditional pooling techniques can sometimes be at odds with fundamental principles of signal processing, specifically the Nyquist sampling theorem.
The Nyquist sampling theorem states that a continuous signal can be completely represented by its samples and fully reconstructed if it is sampled at least twice as fast as its highest frequency component. Traditional pooling methods, including max-pooling and strided convolutions, often don't adhere to this theorem. They perform what is known as 'aliasing,' which can cause a loss of information and make the CNN sensitive to small shifts or translations in the input.
Research from R. Zhang's 2019 ICML paper titled 'Making Convolutional Networks Shift-Invariant Again' sheds light on this issue. It points out that simple anti-aliasing techniques, which involve low-pass filtering before downsampling, aren't often used in modern CNN architectures, mostly because they've been found to degrade performance when not implemented carefully.
However, Zhang shows that when integrated correctly, anti-aliasing techniques can co-exist with traditional downsampling methods like max-pooling and strided convolutions. This not only improves the model's accuracy on benchmarks like ImageNet but also enhances its generalization capabilities, making it more stable and robust to input variations and corruptions.
So, the takeaway here is that classical signal processing techniques, when applied judiciously, can rectify some of the shortcomings of modern CNNs, making them more robust and accurate. This suggests that the field might benefit from revisiting some of these foundational principles.

frequencies in images

- https://www.youtube.com/watch?v=js4bLBYtJwY
- https://medium.com/@shashimalsenarath.computer-vision-d179b8c0f723





Deep Learning – Bernhard Kainz

When we discuss the concept of frequency in images, it's helpful to first consider how frequency works in the context of sound waves. In sound, frequency refers to the rate at which a sound wave oscillates. High-pitched sounds, like that of a violin, have high frequencies, whereas lower-pitched sounds, like the drum, have low frequencies.

But what does frequency mean when it comes to images? In essence, frequency in images is a measure of how quickly the intensity values or brightness of the image change from one point to another. Just as in sound, 'high frequency' in images implies rapid changes in intensity or color over a small spatial range. This can manifest as edges, textures, or intricate details within the image.

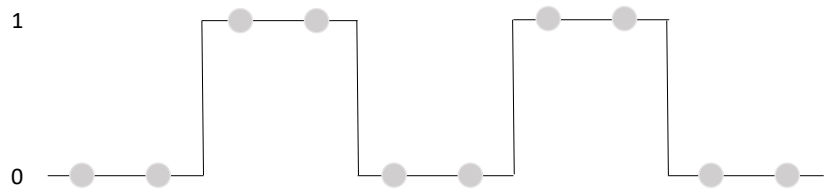
On the other hand, low-frequency elements in images represent areas where the intensity or color changes more slowly and gradually. These could be large uniform regions or smooth gradients in the image.

To put it simply, when we say an image is 'high-frequency,' we're usually talking about an image with lots of edges, textures, and fine details. When we say an image is 'low-frequency,' it typically has fewer details and more uniform or smoothly varying regions.

Understanding frequency in images is crucial when we talk about convolutions and pooling in CNNs, as different filters can capture different frequency components of the image, and certain operations can preserve or destroy these components.

Simple example

- Max-pooling breaks shift equivariance

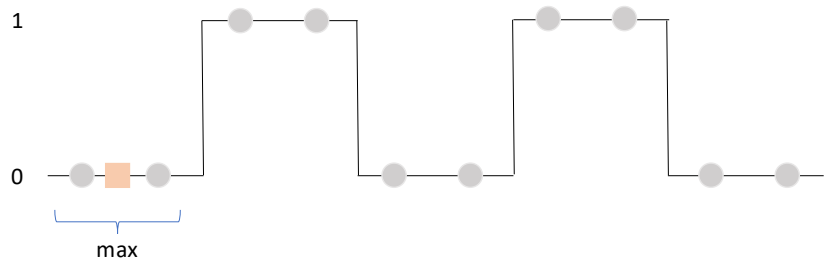


https://www.youtube.com/watch?v=eZa56DqXTHg

Simple toy signal. It goes from 0 to 1 and back

Simple example

- Max-pooling breaks shift equivariance

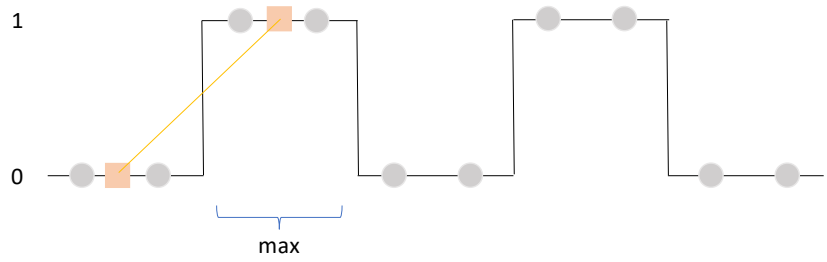


https://www.youtube.com/watch?v=eZa56DqXTHg

Let's max pool it together
0 and 0 is 0

Simple example

- Max-pooling breaks shift equivariance

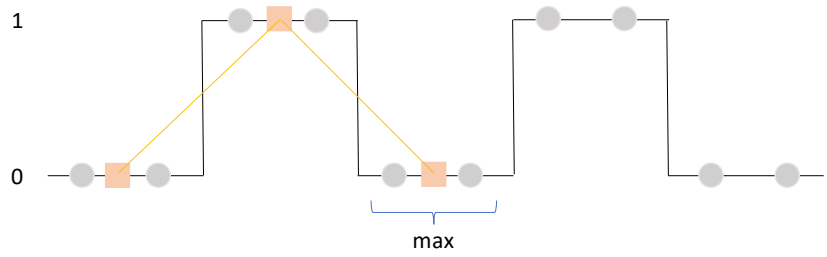


https://www.youtube.com/watch?v=eZa56DqXTHg

Let's max pool it together
0 and 0 is 0

Simple example

- Max-pooling breaks shift equivariance

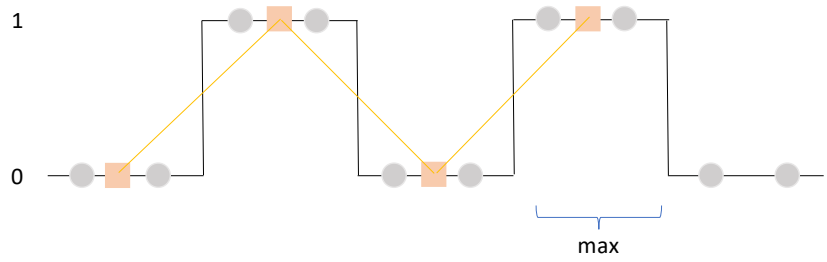


https://www.youtube.com/watch?v=eZa56DqXTHg

Let's max pool it together
0 and 0 is 0

Simple example

- Max-pooling breaks shift equivariance

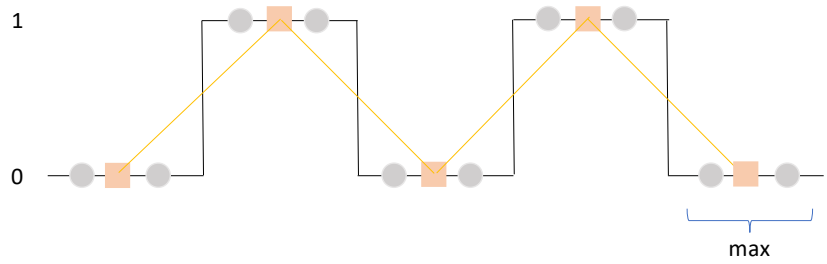


https://www.youtube.com/watch?v=eZa56DqXTHg

Let's max pool it together
0 and 0 is 0

Simple example

- Max-pooling breaks shift equivariance

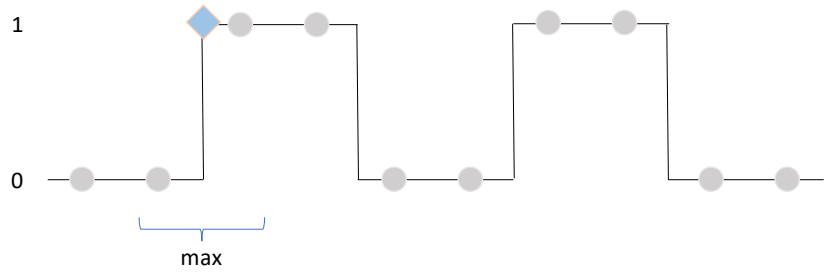


https://www.youtube.com/watch?v=eZa56DqXTHg

Let's max pool it together
0 and 0 is 0

Simple example

- Max-pooling breaks shift equivariance

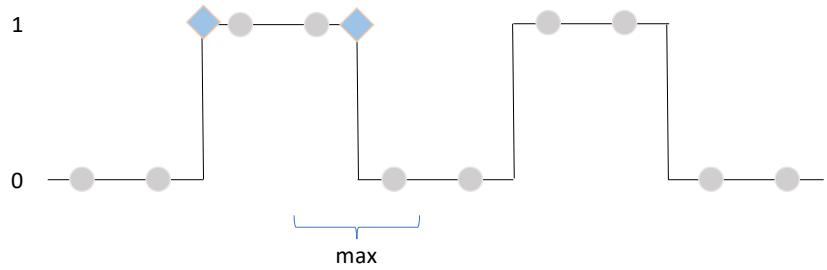


https://www.youtube.com/watch?v=eZa56DqXTHg

Now if we shift the signal by one index we get the max of 0 and 1

Simple example

- Max-pooling breaks shift equivariance

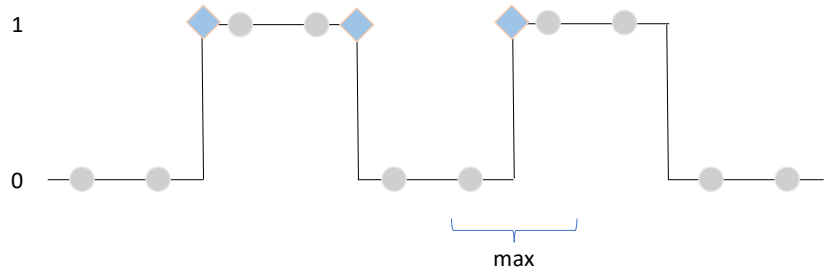


https://www.youtube.com/watch?v=eZa56DqXTHg

Now if we shift the signal by one index we get the max of 0 and 1

Simple example

- Max-pooling breaks shift equivariance

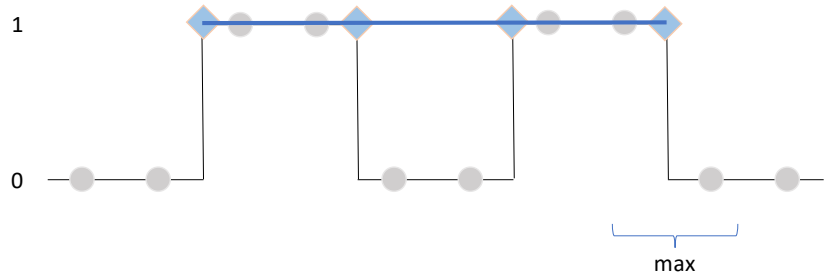


https://www.youtube.com/watch?v=eZa56DqXTHg

Now if we shift the signal by one index we get the max of 0 and 1

Simple example

- Max-pooling breaks shift equivariance



https://www.youtube.com/watch?v=eZa56DqXTHg

Now if we shift the signal by one index we get the max of 0 and 1

Simple example

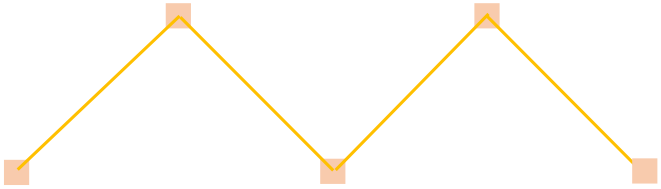
- Max-pooling breaks shift equivariance

https://www.youtube.com/watch?v=eZa56DqXTHg

Now if we shift the signal by one index we get the max of 0 and 1

Simple example

- Max-pooling breaks shift equivariance



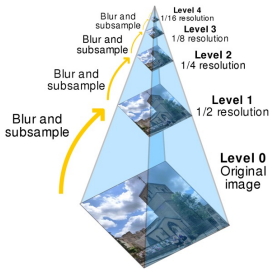
https://www.youtube.com/watch?v=eZa56DqXTHg

Now you would probably agree that these two signals look very different.
What we have done here is that we have broken shift equivariance.

Simple example

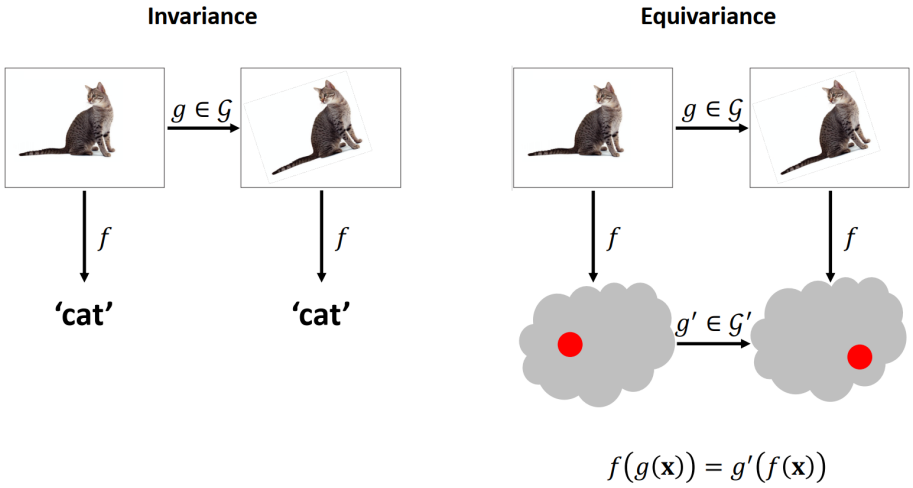
- Max-pooling breaks shift equivariance
- Partial solution: use what you learned about anti-aliasing in Computer Vision: blur and then down sample

R. Zhang.
Making Convolutional Networks Shift-Invariant Again.
In ICML, 2019.



https://www.youtube.com/watch?v=eZa56DqXTHg

Beyond shifts: group equivariance



$$f(g(\mathbf{x})) = g'(f(\mathbf{x}))$$

Deep Learning – Bernhard Kainz

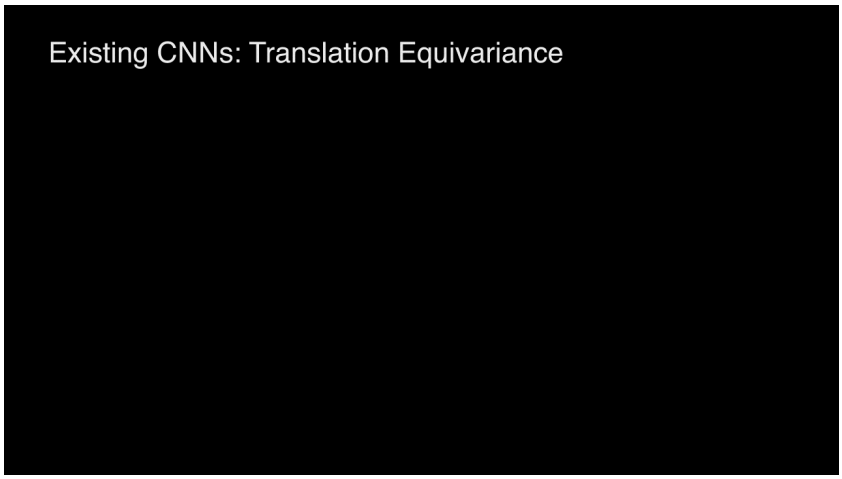
But what about other forms of transformation like rotation? In many cases, especially in computer vision tasks like object recognition or medical imaging, the orientation of an object shouldn't affect our algorithm's ability to recognize it. Unfortunately, standard CNNs are not naturally equivariant to rotations.

Before diving into how we can make CNNs rotation-equivariant, let's talk a bit about group theory. In mathematics, a group is a set of elements combined with an operation that is closed, meaning that combining two elements always results in another element from the same set. Rotations are a simple example of a group. If you rotate an object a bit to the left and then rotate it again, these two rotations can be combined into one effective rotation that achieves the same end result.

This leads us to Harmonic Networks or H-Nets. These are a specialized form of CNNs that are not just equivariant to translation but also to 360-degree rotations. This is achieved by replacing the standard filters in a CNN with what are called 'circular harmonics.' These specially designed filters ensure that a rotation in the input results in a proportionate rotation in the output feature maps, thus giving us rotation-equivariance.

The beauty of H-Nets is that they can be integrated into existing CNN architectures and they are computationally efficient. They've been shown to outperform standard CNNs on tasks like rotated-MNIST and they provide competitive results in other benchmarks.

Rotation invariant CNNs

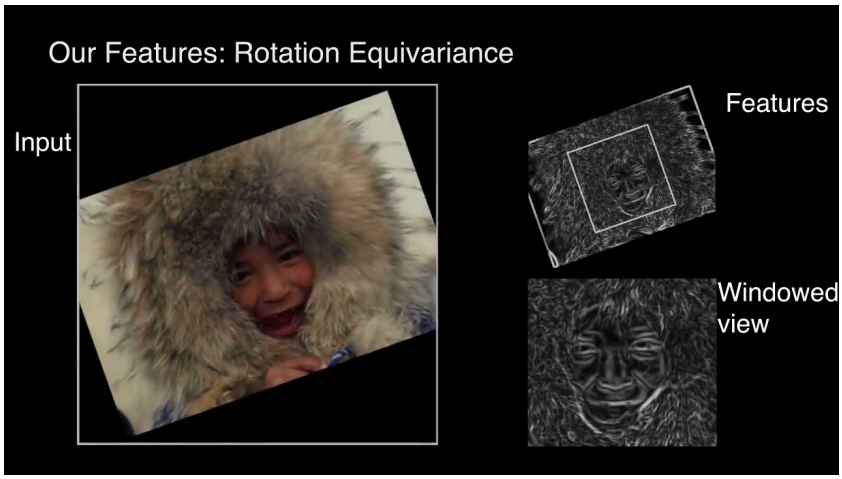


Daniel Worrall et al.: Harmonic Networks: Deep Translation and Rotation Equivariance

https://www.youtube.com/watch?v=goWAFBYOtoU

Achieving group invariance like for example rotation invariant CNNs is reasonable challenging. The paper in this work applied spherical harmonics to achieve this.
If you look at the features maps when rotated the filters in this classification network react differently to different rotations. With harmonics the feature maps stays more or less constant.

Rotation invariant CNNs

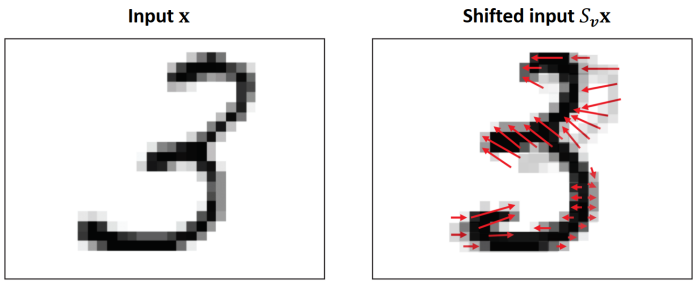


Daniel Worrall et al.: Harmonic Networks: Deep Translation and Rotation Equivariance

https://www.youtube.com/watch?v=goWAFBYOtoU

Achieving group invariance like for example rotation invariant CNNs is reasonable challenging. The paper in this work applied spherical harmonics to achieve this.
If you look at the features maps when rotated the filters in this classification network react differently to different rotations. With harmonics the feature maps stays more or less constant.

Approximate deformation invariance



- **'Digit 3 detector'** $f: \mathbb{R}^d \to \mathbb{R}$
- **Warp operator** $D_{\boldsymbol{\tau}}: \mathbb{R}^d \to \mathbb{R}^d$ warping the image by field $\boldsymbol{\tau}$

Deformation invariance: $f(\mathbf{x}) \approx f(D_{\boldsymbol{\tau}}\mathbf{x})$

Let's dive deeper into the concept of 'equivariance,' which goes beyond just shift-invariance. Our goal here is to appreciate the extent to which CNNs can be adapted to capture a wider variety of transformations. We've discussed translation and rotation, but what about more complex changes like deformations?

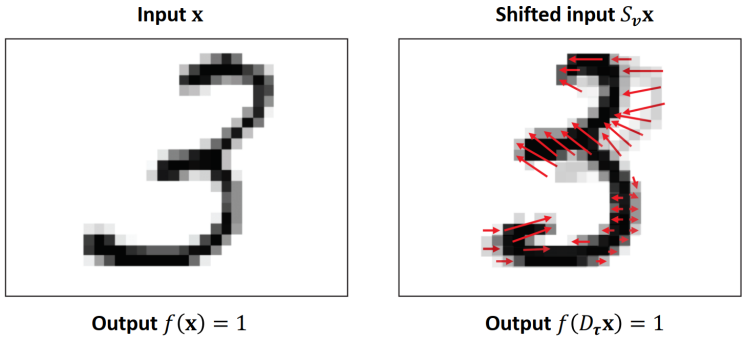
Imagine a canonical representation of the number '3' from the MNIST dataset. Now consider all the possible ways people might write this number. Some might elongate the curves; others might write it more compressed. These are what we term as 'deformations,' and they can be represented mathematically as warping operations.

In a more technical sense, a warping operation applies a smooth deformation field to the pixels of an image, shifting them slightly to create a new yet similar arrangement. It's these small, localized shifts that CNNs are particularly good at handling. This explains why CNNs excel at tasks like handwritten digit classification.

In essence, CNNs are not just invariant to large-scale, easily-defined transformations like shifts and rotations; they are also approximately invariant to these subtle, complex deformations. This adaptability makes CNNs a powerful tool for a wide array of image recognition tasks, from medical imaging to autonomous driving.

The takeaway here is that while CNNs were originally designed with translation invariance in mind, their utility extends far beyond, allowing them to capture and generalize well even in the presence of more complex deformations.

Approximate deformation invariance



- **'Digit 3 detector'** $f: \mathbb{R}^d \to \mathbb{R}$
- **Warp operator** $D_{\boldsymbol{\tau}}: \mathbb{R}^d \to \mathbb{R}^d$ warping the image by field $\boldsymbol{\tau}$

$$\|f(\mathbf{x}) - f(D_{\boldsymbol{\tau}}\mathbf{x})\| \approx \|\nabla \boldsymbol{\tau}\|$$

CNNs can adapt to deformations or warping operations through a vector field τ, which describes these shifts at a local level. However, one crucial factor that determines the success of CNNs in dealing with such deformations is how this vector field τ differs from a constant shift.

In a constant shift, every pixel moves by the same amount in the same direction. It's like moving a photograph sideways; every part of it moves uniformly. This is relatively easy for a CNN to handle due to its inherent translation invariance. However, in real-world scenarios, the shifts are often non-uniform; some pixels might move a lot, while others might barely move. These are described by a variable vector field ?τ.

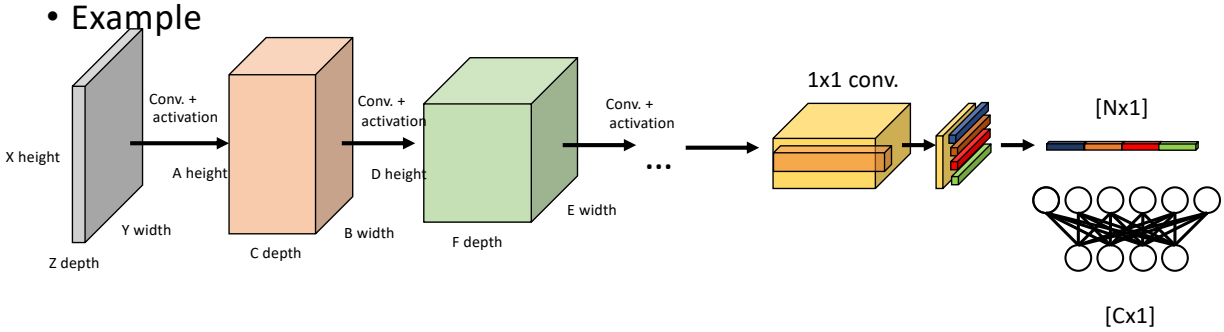
For instance, in the context of handwriting recognition, two instances of the number '3' may have similar overall shapes but slightly different curvatures or thicknesses. In a facial recognition scenario, the same face may appear slightly stretched or compressed due to the camera angle. The difference from a constant shift in these examples is crucial for classification tasks.

The effectiveness of a CNN in these situations hinges on its ability to learn these complex, non-constant shifts and deformations. It's this nuanced adaptability that often separates good CNN models from great ones.

So, the question that model architects are now focusing on is not just how to make CNNs invariant to shifts, but how to make them adapt to a variable field of local shifts, thereby improving their performance and applicability.

Flattening

- Example



Deep Learning – Bernhard Kainz

Flattening layers serve as a connector between convolutional layers and fully connected layers in a Convolutional Neural Network (CNN).

In a CNN, the convolutional and pooling layers usually output a 3D tensor that represents the learned features from the input image.

However, fully connected layers expect a 1D tensor of numbers.

The role of the flattening layer is to bridge this gap by reshaping the 3D tensor into a 1D tensor.

Here's a simple example: Let's say the output of your final pooling layer in a CNN is a tensor of shape (4,4,64), which means it has a height of 4, a width of 4, and a depth (number of feature maps) of 64.

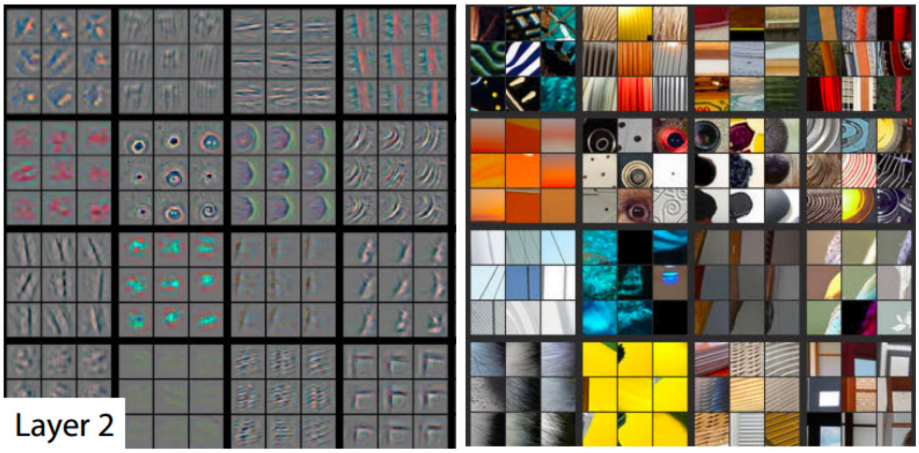
The flattening layer will take this $4\times4\times64$tensor and reshape it into a 1D tensor of shape 1×1024 without altering the actual data.

The flattened data serves as the input to the subsequent fully connected layers (also known as dense layers).

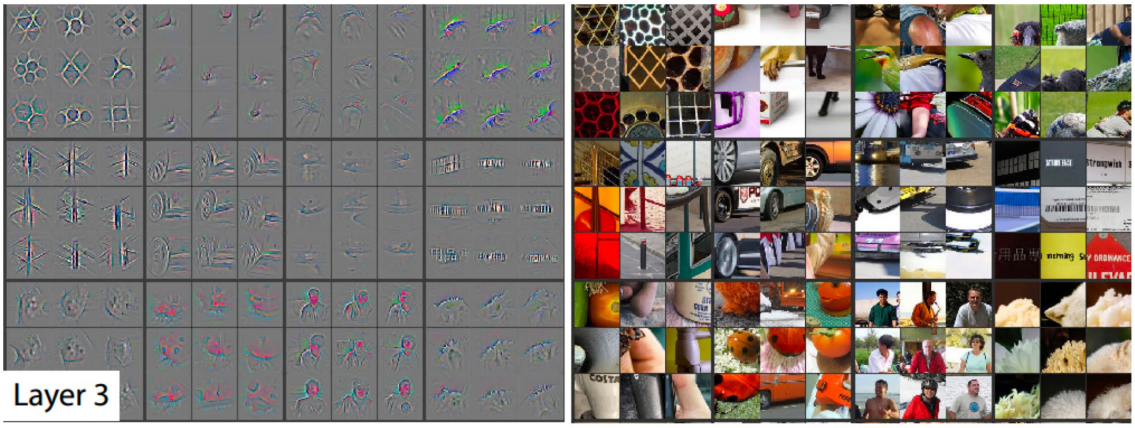
The flattening layer doesn't learn any parameters; it only reformats the data.

This step is crucial for transforming the spatial feature data into a format that can be fed into standard fully connected layers for tasks like classification.

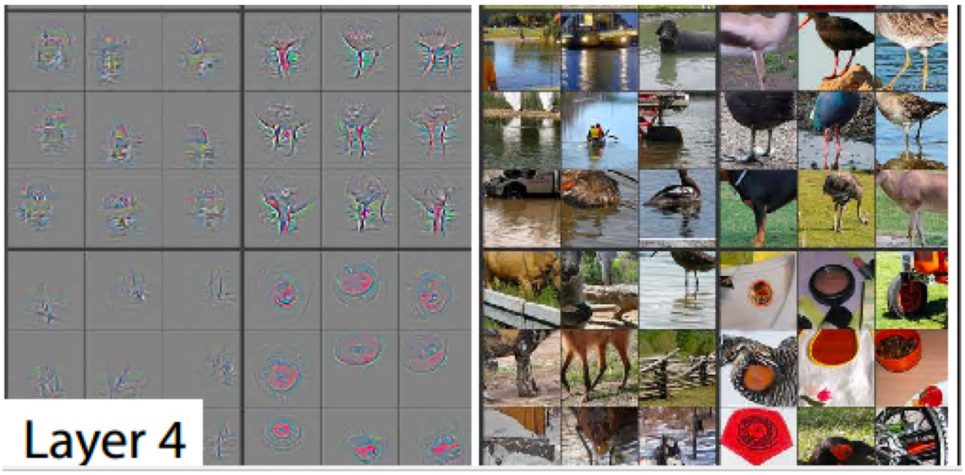
What CNNs learn?



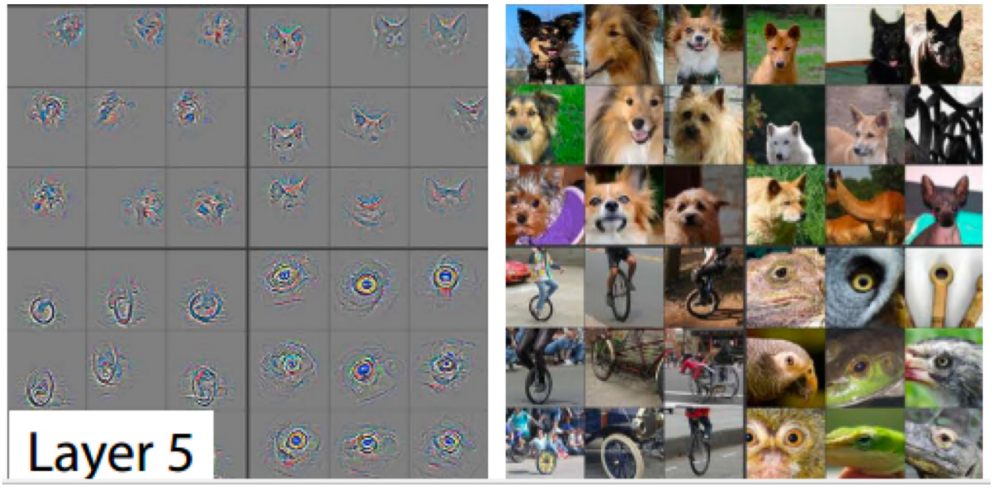
What CNNs learn?



What CNNs learn?

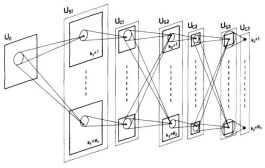


What CNNs learn?



Neocognitron

Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position



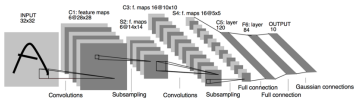
K. Fukushima

Fukushima 1980

Lacks backprop

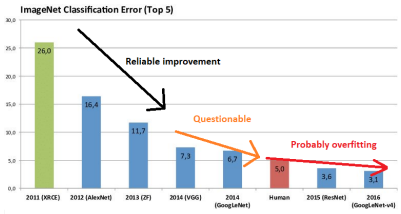
Gradient-Based Learning Applied to Document Recognition

YANN LECUN, MEMBER, IEEE, LÉON BOTTOU, YOSHUA BENGIO, AND PATRICK HAFFNER



LeCun et al. 1998

Adds backprop



Success in 2012!

No GPUs, no success for larger problems…

Historic notes

what do we learn from that?

- a) feature selection is important to build good representations. As we will see, the key of deep learning is to learn this feature selection instead of doing it manually.
- b) finding the right amount of features is key. Too few or too many will have a severe impact on the generalization abilities of your predictor model. Too few is easy too understand but too many requires an intuition about sample sparsity in high-dimensional spaces.
- c) the more features we choose as input the sparser our training samples will be distributed in the feature space. This means that decision boundaries become really tight around the used training samples because they all live close to each other at the boundaries of the space and our model will overfit the training data.

what do we learn from that?

- a) weight sharing reduces the number of parameters from n^2 in a multi-layer perception to a small number, for example 3 as in our experiment or 3 by 3 image filter kernels or similar
- b) these filter kernels can be learned through back propagation exactly in the same way as you would train a multi-layer perception. Each layer may have many filter-kernels, so it will produce many filtered versions of the input with different filter functions.
- c) for real-valued functions, of a continuous or discrete variable, convolution differs from cross-correlation only in that either f(x) or g(x) is reflected about the y-axis; so it is a cross-correlation of f(x) and g(−x), or f(−x) and g(x).

what do we learn from that?

- a) convolutions can massively reduce the computational complexity of neural networks but the real power of CNNs is revealed when priors are implemented and for example spatial structure is preserved. This is also one of the reasons why CNNs have been so successful in Computer Vision
- b) CNNs are pipeline of learnable filters interleaved with nonlinear activation functions producing d-dimensional feature maps at every stage. Training works like a common neural network: initialise randomly, present exampled from the training database, update the filter weights through backpropagation by propagating the error back through the network.
- c) convolution and pooling can be used to reduce the dimensionality of the input data until it forms a small enough representation space for either traditional machine learning methods for classification or regression or to steer other networks to for example generate a semantic interpretation like a mask of a particular object in the input.