Department of Computing

Imperial College of Science, Technology and Medicine

Convolutional Neural Networks and Curse of Dimensionality

Explains the concept of curse of dimensionality and refresher on CNN architectures

Author: Anton Zhitomirsky

Contents

1 Learning System

1.1 Fundamental components

The learning framework consists of 4 fundamental milestones:

1. Feature Extractor: responsible for capturing the hierarchical patterns in the data - edges, textures, and more complex shapes. It is what allows the network to 'understand' what it's looking at.
2. Task-specific Head: tailored to the problem we're trying to solve. Takes the rich features extracted by the feature extractor and maps them to the specific task at hand.
3. Parameter Optimization: using algorithms like stochastic gradient descent, the model fine-tunes its parameters to minimize a loss function.

1.2 Challenges in image analysis

1. dealing with the sheer complexity and size of image data
2. Traditional neural networks, however, don't have an innate understanding of the spatial hierarchies that are present in image data
3. 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
4. over-fitting - The risk amplifies because of the high number of parameters involved in these models

1.3 Universal Approximator

"The Universal Approximation Theorem tells us that Neural Networks has a kind of universality i.e. no matter what f(x) is, there is a network that can approximately approach the result and do the job! This result holds for any number of inputs and outputs" [4]

Theorem 1.1 (Universal Approximator)**.** Let $\phi(\cdot)$ be a non-constant, bounded[1] and monotonically increasing function. For any $\epsilon > 0$ and any continuous function defined on a compact subset of $\mathbb{R}^m$, there exists an 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:

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

We assume that ϕ is a sensible activation function, like the sigmoid or ReLU.

1.3.1 Problems

1. ϵ can be very large in practice, making the approximation less useful.
2. Curse of dimensionality[2], where the number of computational and spatial complexity increases exponentially with the number of dimensions.

[1] It has been shown that it even works with unbounded
[2] Section 2

1.3.2 Solution

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.

2 Curse of Dimensionality

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

Definition 2.1. As the number of features or dimensions grows, the amount of data we need to generalize accurately grows exponentially.

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

We also find that as the number of dimensions increases, the performance of a model increases and falls after a point [5].

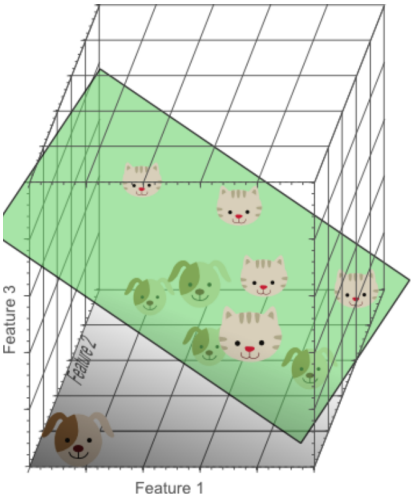


Figure 5. The more features we use, the higher the likelihood that we can successfully separate the classes perfectly.

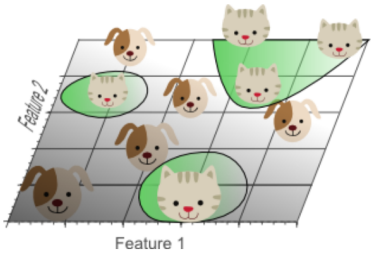


Figure 6. Using too many features results in overfitting. The classifier starts learning exceptions that are specific to the training data and do not generalize well when new data is encountered.

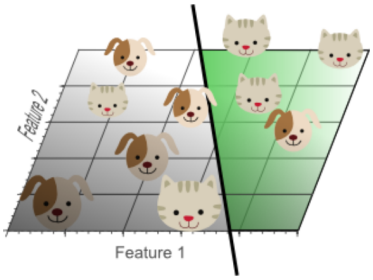


Figure 7. Although the training data is not classified perfectly, this classifier achieves better results on unseen data than the one from figure 5.

Figure 1: Demonstration of overfitting when introducing more features [5]

Adding the third dimension in Figure 1 to obtain perfect classification results, simply corresponds to using a complicated non-linear classifier in the lower dimensional feature space. If we would keep adding features, the dimensionality of the feature space grows, and becomes sparser and sparser. Due to this sparsity, it becomes much more easy to find a separable hyperplane because the likelihood that a training sample lies on the wrong side of the best hyperplane becomes infinitely small when the number of features becomes infinitely large. This concept is called overfitting and is a direct result of the curse of dimensionality.

2.1 Sample Explosion

If we have a function f as in Theorem 1.1 and we want to approximate this function with a specific level of accuracy, denoted as ϵ, then the number of sampels required to achieve this level of accuracy ϵ grows exponentially with dimension d as in Definition 2.1.

2.2 Sparseness

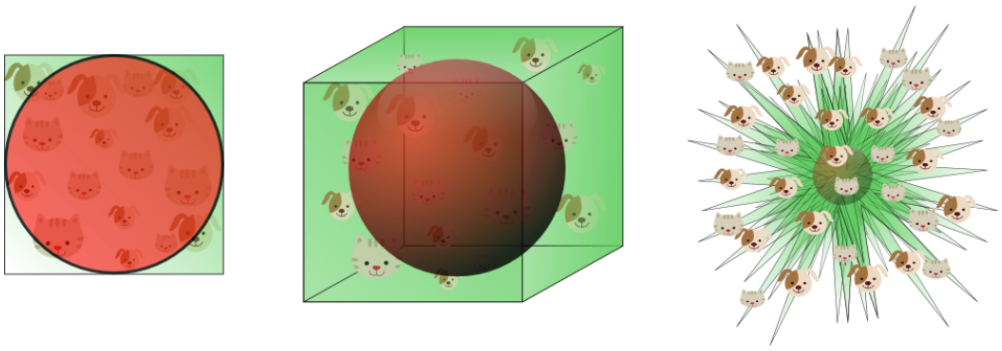


Figure 11. As the dimensionality increases, a larger percentage of the training data resides in the corners of the feature space.

Figure 2: Sparseness of data as you add more dimensions. The data points sit around the edges, rather than in the center [5]

The more features we use, the more sparse the data becomes such that accurate estimation of the classifier's parameters (i.e. its decision boundaries) becomes more difficult. Another effect of the curse of dimensionality, is that this sparseness is not uniformly distributed over the search space. In fact, data around the origin (at the center of the hypercube) is much more sparse than data in the corners of the search space. [5].

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 8samples in 2D, and potentially 1,000 in 3D. The sample requirement explodes as dimensions increase, making the task computationally burdensome.

2.2.1 Math

Intuition taken from slides and [3].

1. Suppose you have a n-dimensional shape. It is described by a set of n vectors. If we want to create a proportionately smaller vector by factor α then we multiply by α.
2. Comparing the volume in the case of the 3-dimensional shape:

$$V^3_{original} = Width \times Height \times Length \tag{2}$$
$$V^3_{interior} = \alpha Width \times \alpha Height \times \alpha Length \tag{3}$$
$$= \alpha^3 V_{original} \Rightarrow V^n_{interior} = \alpha^n V_{original} \tag{4}$$
$$V^3_{rind} = (1-\alpha)Width \times (1-\alpha)Height \times (1-\alpha)Length \tag{5}$$
$$= (1-\alpha^3)V_{original} \Rightarrow V^n_{rind} = (1-\alpha^n)V_{original} \tag{6}$$

3. The volume of the rind relative to the original volume therefore is

$$\frac{V_{rind}}{V_{original}} = 1 - \alpha^n \tag{7}$$

4. We can define the rate of change of α relative to the ratio above as

$$\frac{d(1-\alpha^n)}{d\alpha} = -n\alpha^{n-1} \iff d(1-\alpha^n) = -n\alpha^{n-1}d\alpha \tag{8}$$

5. From Equation 8 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 (when $\alpha = 1$ and $d\alpha < 0$ then $d(1-\alpha^n) = n|d\alpha|$)
6. in higher dimensions, relatively tiny changes in distance translate to much larger changes in volume.
7. 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.

We can find, for the general case, what the volume ratio would be to find 50% of the data as follows:

1. $\alpha^n = 0.5$ then $\alpha = 2^{-1/n} \approx 1 - 0.7/n$
2. From this, we have a formula to see how much we ought to shrink the object by such that 50% of the data will be split amongst the rind and internal volume.
3. E.g. in the 2-d case: $\alpha = 1 - 0.35$. Therefore, from the outer edges, $1 - \alpha = 0.35$ (We do $1 - \alpha$ as we needed to in Equation 5).

   $0.35/2$ describes 'cutting from both sides' so 0.18 of its diameter from the boundary will contain 50% of the volume.
4. For large dimensions, the half-length is very close to 1: $n = 350$ gives 98% - thus expect half of any 350-dimensional shape's volume to lie within 1% of its diameter from its boundary.

Therefore, 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 [3].

2.3 Practical Meaning

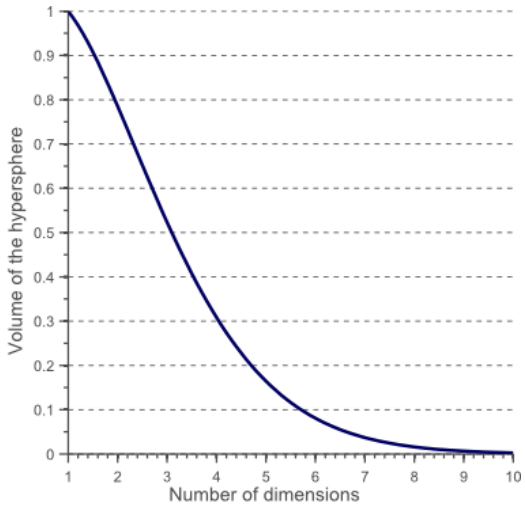
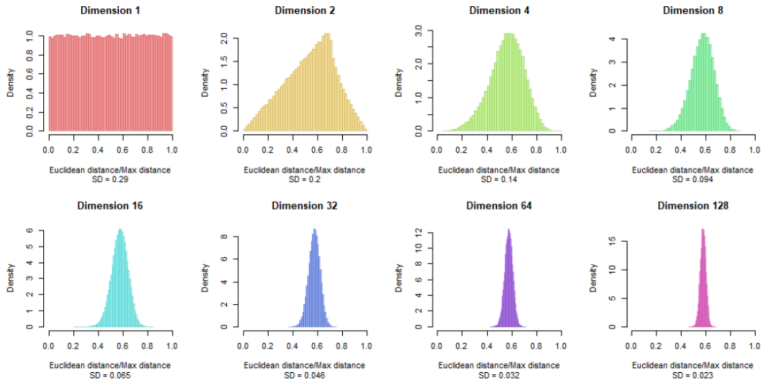


Figure 3: As you increase dimensions the total volume of 'useful' data is way smaller

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.

Infact, as shown in Figure 2 the higher dimensions you have the higher probability that a data-point will sit in its own distinct corner in the hypercube. The distance between the corner will be equal. If we roll the high dimensional enclosing hypercubes into a n-dimensional torus, and we plot the distance between samples, we see that distances between points become approximately equal.



3 Invariance and Equivariance

Definition 3.1 (Shift Invariance)**.** describes a system's unchanging response when the input is shifted. For example, some classifier f and shift operator S_v, $f(\boldsymbol{x}) = f(S_v\boldsymbol{x})$

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.

Shift invariance is beneficial because it allows models to generalize better from their training data to new unseen data.

Definition 3.2 (shift Equivariance)**.** applying the shift operator after the function yields the same results as applying the function after the shift. i.e. $S_v \circ f(\boldsymbol{x}) = f(\boldsymbol{x}) \circ S_v$

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.

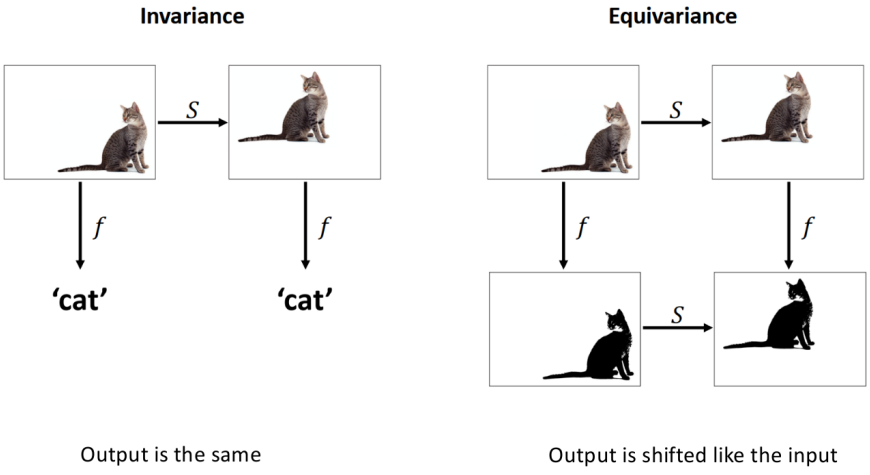


Figure 4: Equivariance vs Invariance

4 Inductive Bias/assumptions

4.1 Translation invariance

A shift in the input should simply lead to a shift in the hidden representation in a predictable manner. The essence of what is being represented should not change simply because its location in the input space has changed.

4.2 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.

4.3 Practical Application

We can use a sliding window and come up with a correlation metric to see if a patch is on an image. The first principle tells us it doesn't matter where in the image the patch is located, the second means that operating under the assumption that lal the information needed to identify the person is found in a local neighborhood of pixels.

5 Convolutions

5.1 Fully connected neural network

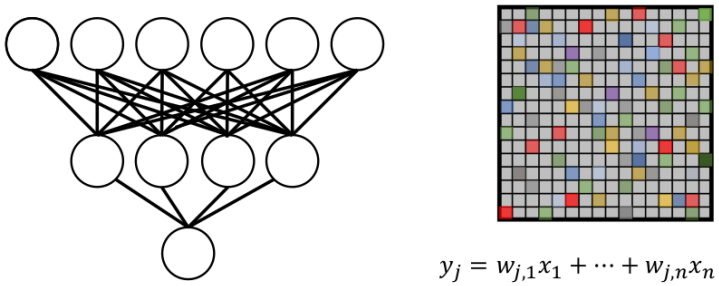


Figure 5: Fully convolutional neural network

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 is flexible, yet overfits and is computationally expensive.

For each node, the output would be a linear combination of all nodes in the layer before it. In an $n \times n$ matrix, then for a standard high-resolution image of 36 million elements, this will be too many parameters.

5.2 Sparsely Connected neural networks

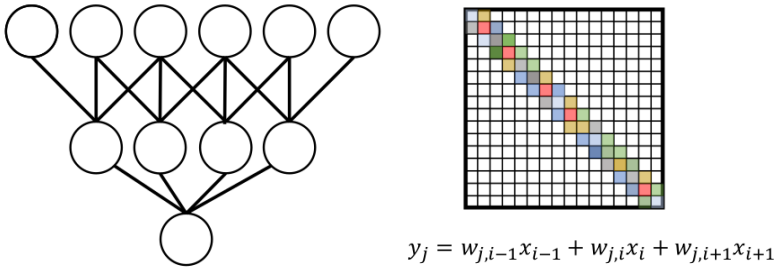


Figure 6: Sparsely convolutional neural network

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. This reduces the amount parameters greatly, if we choose to connect to k neurons, then size is $k \times n$.

5.3 Weight Sharing neural networks

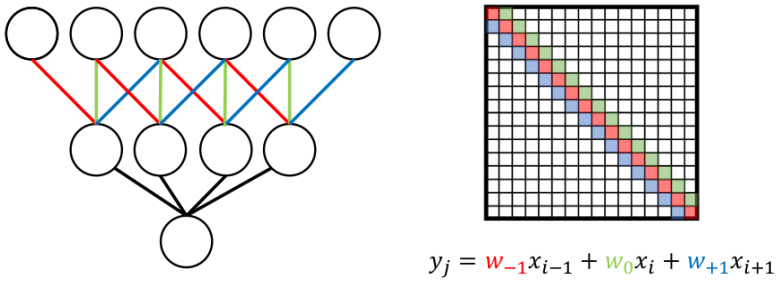


Figure 7: Weight sharing convolutional neural network

The same weights w_{-1}, w_0, w_{+1} are reused for different input neurons. This is what we refer to as 'weight sharing.' 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. 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.

5.4 Convolution vs 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.

Definition 5.1 (Convolution (continuous))**.**

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \quad for f, g : [0, \infty] \to \mathbb{R} \tag{9}$$

This video has a great explanation [1]

Definition 5.2 (Convolution (discrete))**.**

$$s_t = \sum_{a=-\infty}^{+\infty} u_a w_{t-a} \tag{10}$$

Definition 5.3 (Correlation)**.**

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t+\tau) d\tau \quad for f, g : [0, \infty] \to \mathbb{R} \tag{11}$$

5.4.1 Commutativity

Definition 5.4 (Commutativity of convolutions)**.**

$$\begin{aligned}
(f * g)(n) & \quad \text{starting point, what is the convoution at position n?} \\
= \sum_{i=-\infty}^{\infty} f(i) \cdot g(n-i) & \quad \text{by convolution rule} \\
= \sum_{j=n-(-\infty)}^{j=n-\infty} f(n-j') \cdot g(j') & \quad \text{substitute } j' = n - i \\
= \sum_{j=\infty}^{-\infty} f(n-j') \cdot g(j') & \quad \text{fix the bounds of the sum} \\
= \sum_{j=-\infty}^{\infty} f(n-j') \cdot g(j') & \quad \text{commutivity of addition} \\
= (g * f)(n) & \quad \text{apply convolution rule}
\end{aligned}$$

5.4.2 Associativity

Definition 5.5 (Associativity of convolutions)**.**

$$\begin{aligned}
((f * g) * h)(n) & \quad \text{starting point, what is the convoution at position n?} \\
= \sum_{i=-\infty}^{\infty} (f * g)(i) \cdot h(n-i) & \quad \text{by convolution rule} \\
= \sum_{i=-\infty}^{\infty} (\sum_{j=-\infty}^{\infty} f(j) \cdot g(i-j)) \cdot h[n-i] & \quad \text{applying convolution for } (f * g) \\
= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(j) \cdot g(i-j) \cdot h[n-i] & \quad \text{extracting } \sum \text{ since term } n-i \text{ doesn't rely on } j \\
= \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} f(j) \cdot g(i-j) \cdot h[n-i] & \quad \text{switch sum of } i \text{ and } j \\
= \sum_{j=-\infty}^{\infty} f(j) \cdot \sum_{i'=-\infty}^{\infty} g(i') \cdot h[n-j-i'] & \quad \text{substitute } i' = i - j \text{ and extract } f(j) \\
= \sum_{j=-\infty}^{\infty} f(j) \cdot (g * h)(n-j) & \quad \text{apply convolution rule} \\
= (f \cdot (g * h))(n) & \quad \text{apply convolution rule}
\end{aligned}$$

5.4.3 Distributivity

Definition 5.6 (Distributivity of convolutions)**.**

$$
\begin{aligned}
(f_1 * (f_2 + f_3))(n) & \quad \text{starting point, what is the convoution at position n?} \\
= \sum_{i=-\infty}^{\infty} f_1(i) \cdot (f_2(n-i) + f_3(n-i)) & \quad \text{by convolution rule} \\
= \sum_{i=-\infty}^{\infty} (f_1(i) \cdot f_2(n-i)) + (f_1(i) \cdot f_3(n-i)) & \quad \text{expanding parenthesis} \\
= \sum_{i=-\infty}^{\infty} (f_1(i) \cdot f_2(n-i)) + \sum_{i=-\infty}^{\infty} (f_1(i) \cdot f_3(n-i)) & \quad \text{splitting summation} \\
= (f_1 * f_2 + f_1 * f_3)(n) & \quad \text{apply convolution rule on both sums}
\end{aligned}
$$

5.4.4 Associativity with scalar multiplication

Definition 5.7 (Associativity with scalar multiplication of convolutions)**.**

$$
\begin{aligned}
(a(f_1 * f_2))(n) & \quad \text{starting point, what is the convoution at position n?} \\
= a \sum_{i=-\infty}^{\infty} (f_1)(i) \cdot (f_2)(n-i) & \quad \text{by convolution rule} \\
= \sum_{i=-\infty}^{\infty} a(f_1)(i) \cdot (f_2)(n-i) & \quad \text{push through } a \\
= (af_1) * f_2 & \quad \text{apply convolution rule} \\
= \sum_{i=-\infty}^{\infty} a(f_1)(i) \cdot (f_2)(n-i) & \quad \text{by convolution rule} \\
= \sum_{i=-\infty}^{\infty} (f_1)(i) \cdot a(f_2)(n-i) & \quad \text{by commutivity of scalar multiplication} \\
= f_1 * (af_2) & \quad \text{apply convolution rule}
\end{aligned}
$$

5.5 Why convolutions for pattern-matching?

5.5.1 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.

5.5.2 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.

5.5.3 Convolution vs Correlation

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 back propagation. Thus, for the purpose of training neural networks, the distinction between the two becomes largely irrelevant.

5.5.4 Implementation

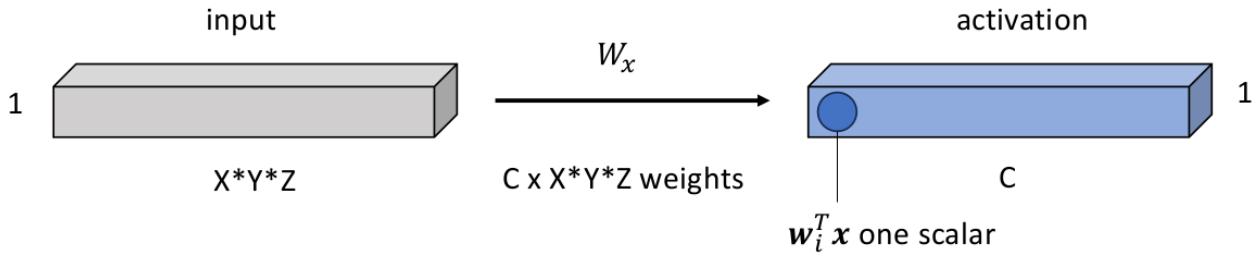
In deep learning frameworks like TensorFlow or PyTorch, the operation performed in the convolutional layers is actually cross-correlation. However, they still use the term “convolution” due to convention.

5.6 Practical applications

Edge Detection, Sharpening, Gaussian Blur

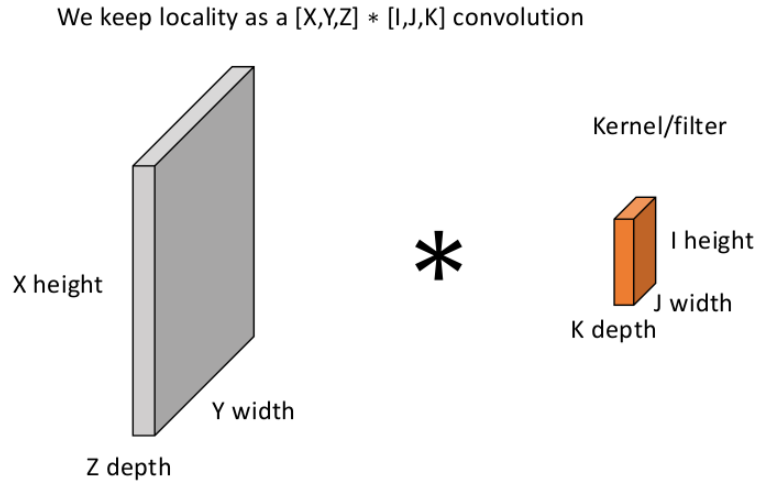
6 CNN

6.1 Input Tensor



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.

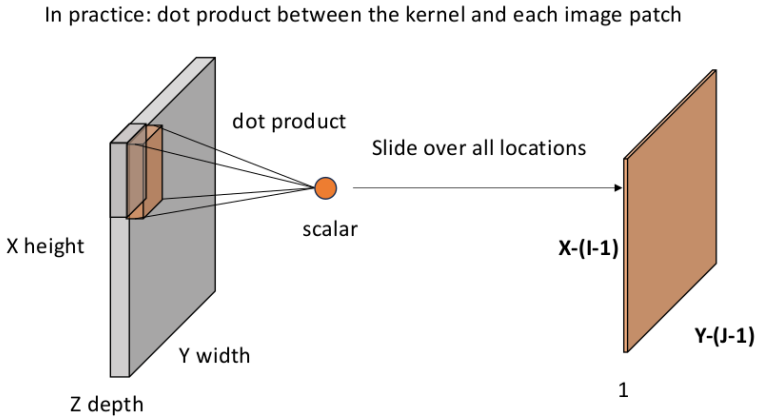
However, we have certain priors about image data that make this approach sub-optimal. Let’s dive into these priors starting with the first principle: translation invariance. What we mean by this is that if you shift an object in an image, the intrinsic characteristics of the object don’t change. Therefore, 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.



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.

6.2 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.

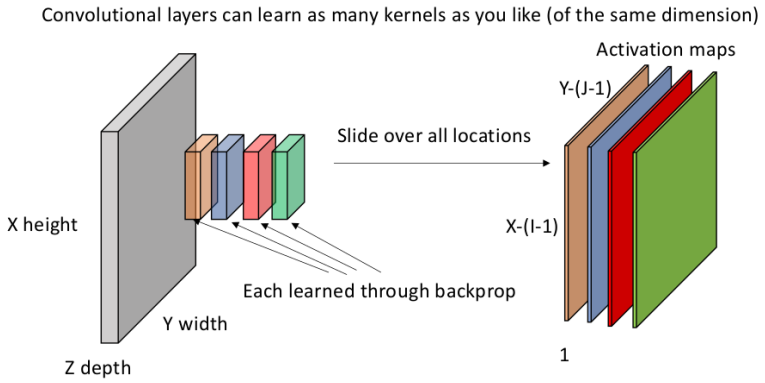


This dot product is essentially a weighted sum of the pixel values in the image patch, where the weights are determined by the kernel. The output of each of these dot products forms a single pixel in the resulting feature map. This method is highly efficient and is one of the reasons why CNNs are so 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.

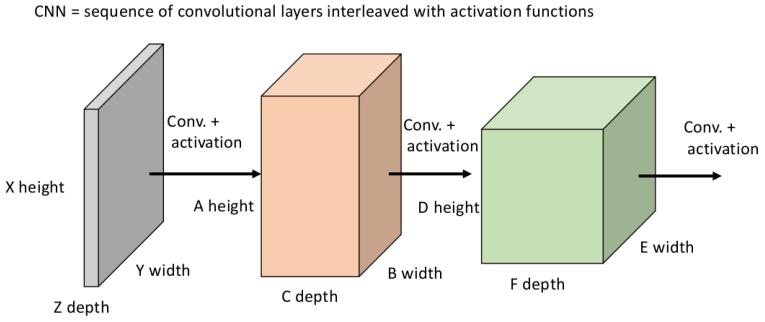
To maintain the size of the image you can use zero-padding. Zero padding is the practice of adding zeros around the edge of the input image before applying the convolution operation.

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.



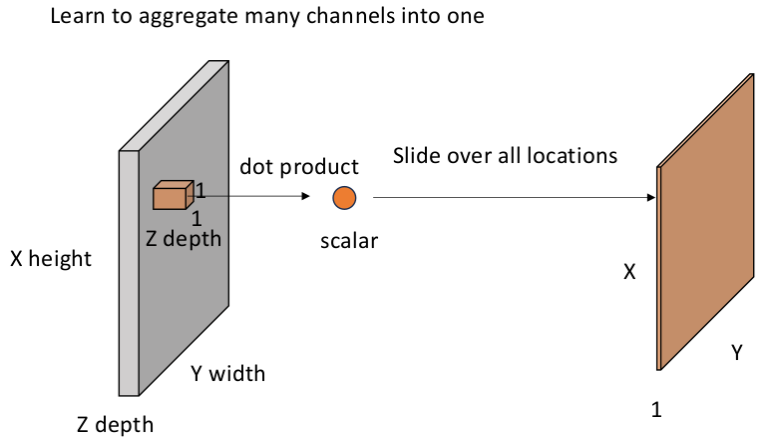
convolutional layers can learn multiple filters, or kernels, at the same time, and all of them are of the same dimension. Why is this important? Each kernel captures a different feature or pattern in the input data. For example, one might specialize in detecting horizontal edges, while another might focus on capturing color gradients. Because these kernels are learned through back propagation, 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



At its core, a CNN is essentially a sequence of convolutional layers, each followed by an activation function.

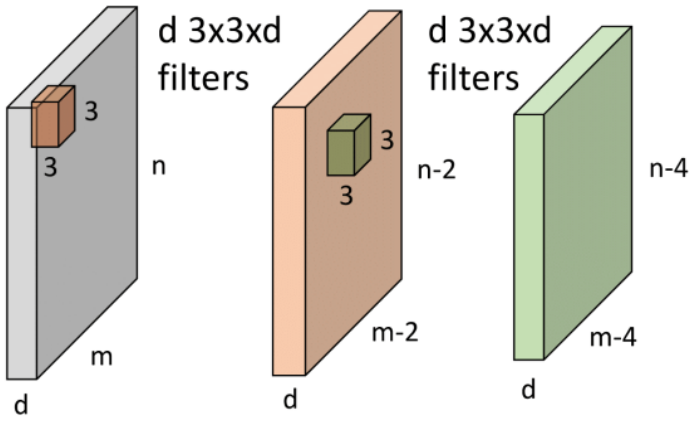
Each filter is defined by its dimensions: I, J, K. For a single filter, total number of parameters are $I \times J \times K + 1(bias)$. The bias term is crucial because it allows the filter to have some flexibility, effectively shifting the activation function to better fit the data.

6.3 1×1 convolution



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.

6.4 Factorized Convolutions



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.

6.5 Separable Convolutions

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.

6.6 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; Permutation-invariant aggregation+downsampling (typically max or avg)

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.

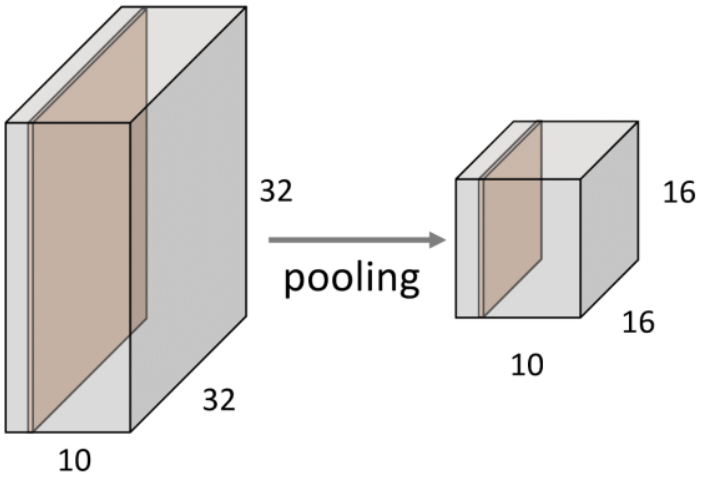


Figure 8: Applied to each channel separately

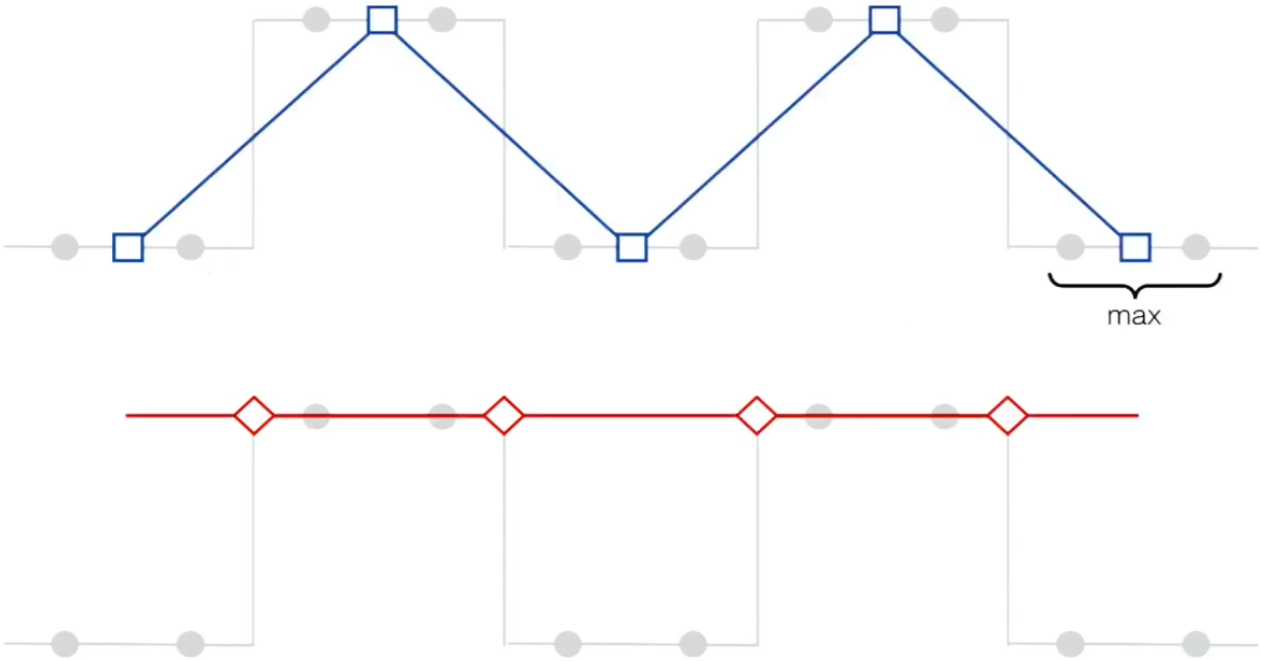
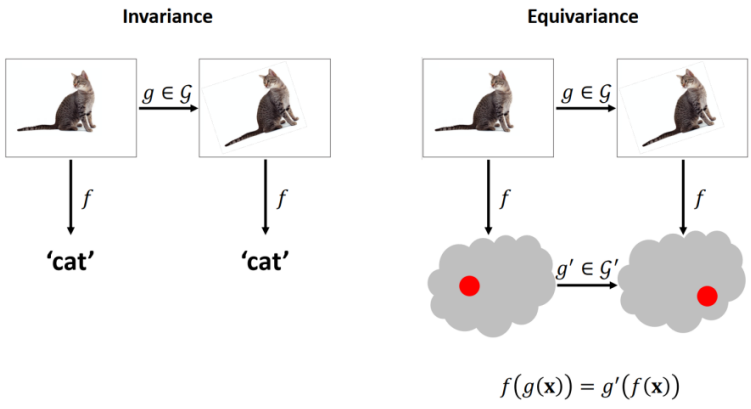


Figure 9: Max-pooling breaks shift equivariance

6.6.1 Max-pooling breaks shift-equivariance [2]

Partial solution: use what you learned about anti-aliasing in Computer Vision: blur and then down sample

6.7 Rotation?



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.

6.8 Approximate Deformation Invariance?

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.

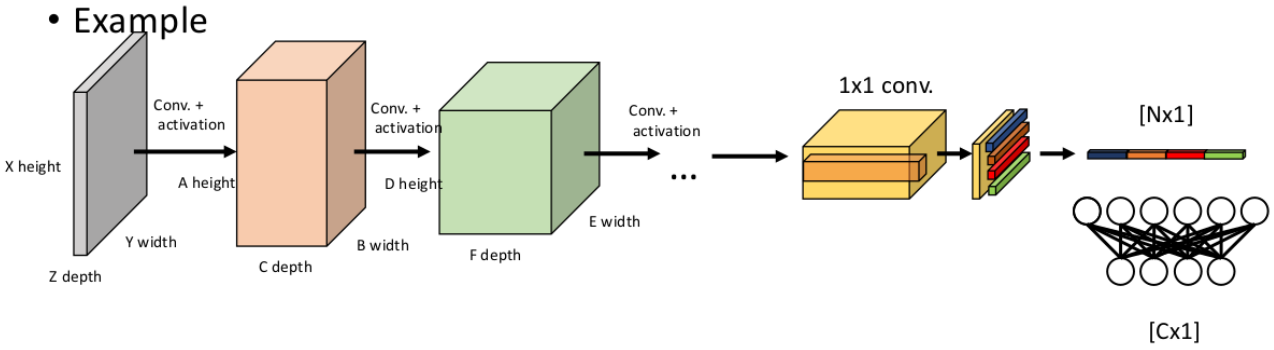
Digit 3 detector: $f : \mathbb{R}^d \to \mathbb{R}$

Wrap operator: $D_\tau : \mathbb{R}^d \to \mathbb{R}^d$ wraping the image field by τ

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

6.9 Flatten Layers

This serves as a bridge between the convolutional layers and the fully connected layers, essentially converting 2D feature maps into 1D vectors.



Flattening layers serve as a connector between convolutional layers and fully connected layers in a Convolutional Neural Network (CNN). The flattening layer doesn't learn any parameters; it only reformats the data.

6.10 Dimensions

After padding and stride, the dimension of the image goes to:

$$o = \lfloor \frac{i + 2p - k - (k-1)(d-1)}{s} \rfloor + 1 \tag{12}$$

Where

- i is the input size
- p is the padding
- k is the convolutional layer size
- d is the dilation
- s is the stride

7 Conclusion

1. 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.
2. 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.
3. 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.
4. 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
5. 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.
6. 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 $g(x)$.
7. 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
8. 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.
9. 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.

References

[1] Three Blue One Brown. *Convolutions — Why X+Y in probability is a beautiful mess — youtube.com*. https://www.youtube.com/watch?v=IaSGqQa5O-M. [Accessed 27-01-2024]. 2023.

[2] *Making Convolutional Networks Shift-Invariant Again (Aug 2019) — youtube.com*. https://www.youtube.com/watch?v=eZa56DqXTHg. [Accessed 27-01-2024].

[3] *Mathematical demonstration of the distance concentration in high dimensions — stats.stackexchange.com*. https://stats.stackexchange.com/questions/451027/mathematical-demonstration-of-the-distance-concentration-in-high-dimensions. [Accessed 27-01-2024].

[4] Milind Sahay. *Neural Networks and the Universal Approximation Theorem — towardsdatascience.com*. https://towardsdatascience.com/neural-networks-and-the-universal-approximation-theorem-8a389a33d30a. [Accessed 27-01-2024].

[5] *The Curse of Dimensionality in Classification — visiondummy.com*. https://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/. [Accessed 27-01-2024].