Machine Learning for High Energy Physics

Introduction to Deep Learning

Prof. Dr. Edmar Egidio Purcino de Souza

15th International Doctorate Network in Particle Physics, Astrophysics, and Cosmology - IDPASC - Natal/2026

Federal University of Bahia
edmar.egidio@ufba.br / edmar.egidio@cern.ch

Introduction to Deep Learning

Introduction to Deep Learning

Introduction

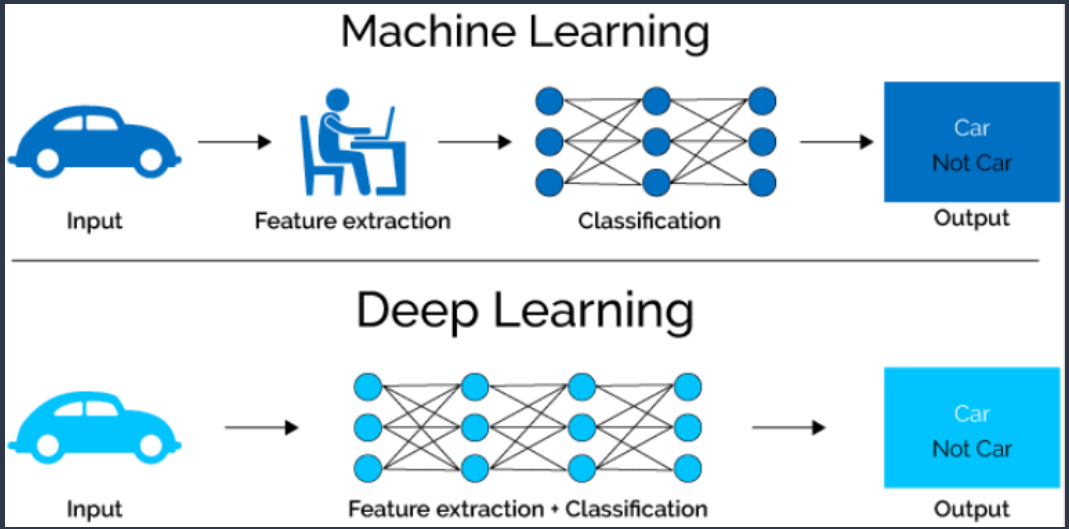
Deep Learning (Goodfellow et al., 2016)
Deep learning is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data using a deep graph with multiple processing layers composed of various linear and non-linear transformations.

Deep Learning (Hinton, G. et al., 2015)
Representation learning is a set of methods that allows a machine to be fed with raw data and automatically discover the representations needed for detection or classification. Deep learning methods are representation learning methods with multiple levels of representation [...]

Introduction

The Feature Extraction Problem:

- Conventional Neural Networks: the commonly adopted solution is feature extraction;
- Examples in images: PCA, Texture features;
- Problems: The relevant features for classification are not always extracted;
- It is context-dependent; It often requires an expert to select features for a specific domain.


Machine Learning
Input
Feature extraction
Classification
Car
Not Car
Output
Deep Learning
Input
Feature extraction + Classification
Car
Not Car
Output

Introduction - Vanishing Gradient Problem

- Neural networks with many layers present successive transformations of the input information;
- They transform the representation at one level, starting from the input layer, into a representation at a higher and slightly more abstract level;
- Problem: **loss of gradient information**. Using Backpropagation, the gradient information is propagated backward. However, the further it gets from the output layer, the more the gradient information decreases;
- This problem, combined with the growth in the number of adjustable parameters (weights), is the main reason why conventional neural networks use few layers;
- The problem is even worse when traditional activation functions like sigmoid and hyperbolic tangent are used;
- The use of the chain rule has the effect of multiplying several small numbers produced by the activation functions, causing the gradient to decrease exponentially.

Convolutional Neural Networks

Convolutional Neural Networks

CNNs are networks that use the concept of Deep Learning:

- Representation learning methods: They allow a machine, when fed with raw data, to automatically discover the best representations for detection or classification;
- It uses representation learning methods with multiple levels of representation. They are composed of simple non-linear modules that transform the representation into a slightly more abstract level;
- With the composition of such simple transformations, very complex functions can be estimated;
- For classification tasks, higher layers of representation amplify aspects of the inputs that are important for discrimination and suppress variations that are irrelevant.

Convolutional Neural Networks

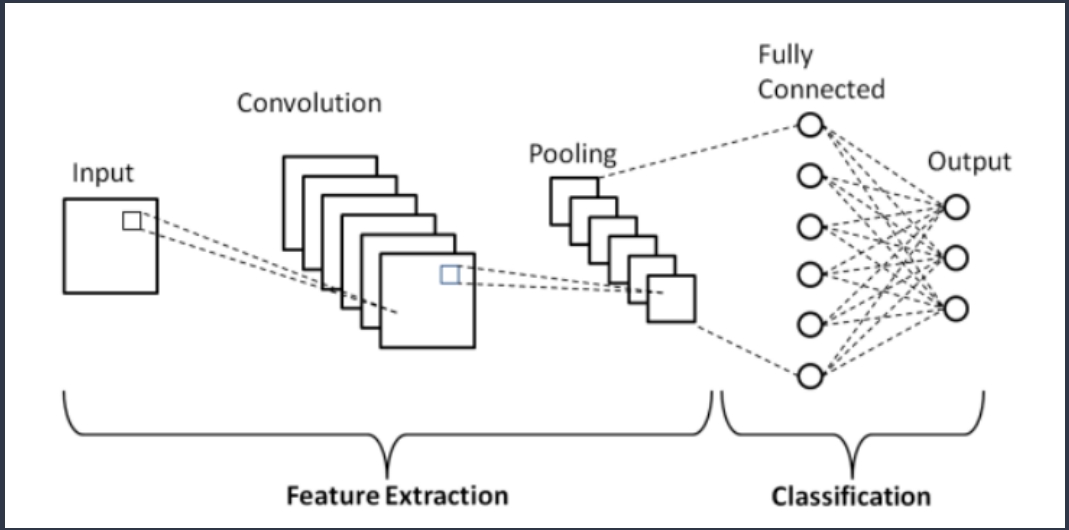


Figure 1: Example of a CNN architecture.

Convolutional Neural Networks

Convolutional networks employ the convolution operation instead of standard matrix multiplication (inherent to a perceptron-type layer) in at least one of their layers (GOODFELLOW ET AL., 2016).

- This operation, which is linear, allows the exploitation of information in structured forms over time (like time series) or space (like images).

The convolution between two sequences can be defined as:

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)w(n-k) = (x * w)(n), \tag{1}$$

Convolutional Neural Networks

Considering a two-dimensional image I used as input, a two-dimensional *kernel*[1] K can be used:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2}$$

And since convolution is commutative:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{3}$$

[1] kernel

Convolutional Neural Networks

The commutative property of convolution arises because the *kernel* has been flipped relative to the input, in the sense that as m increases, the index in the input increases, but the index in the *kernel* decreases.

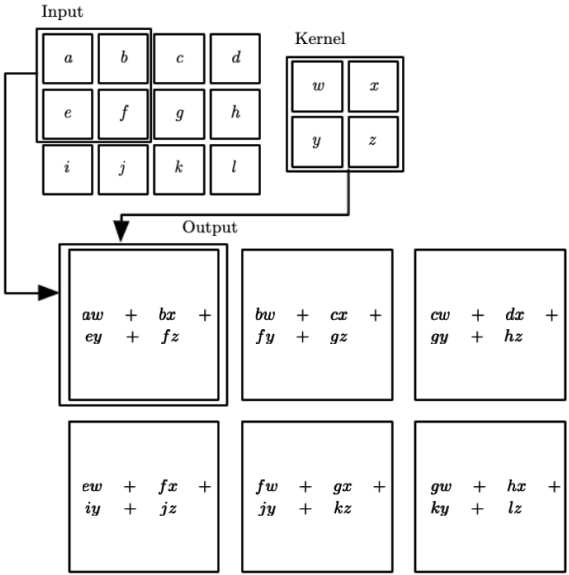
- The only reason to flip the *kernel* is to obtain the commutative property.

In practice, convolutional neural networks implement a function called cross-correlation, which is the same as convolution but without flipping the *kernel*:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{4}$$

- Many machine learning libraries implement cross-correlation but call it convolution.

Convolutional Neural Networks



Key Concepts

- **Parameter Sharing**: The same weights can be used for different parts of the input signals.
- **Sparse Connectivity**: A single element in the feature map is connected to only a small set of pixels. This is very different from connecting to the entire input image, as in the case of multi-layer perceptrons.
- **High Number of Layers**: A high number of layers can combine extracted local features with global features of the input signal set.

CNNs - An Example Architecture

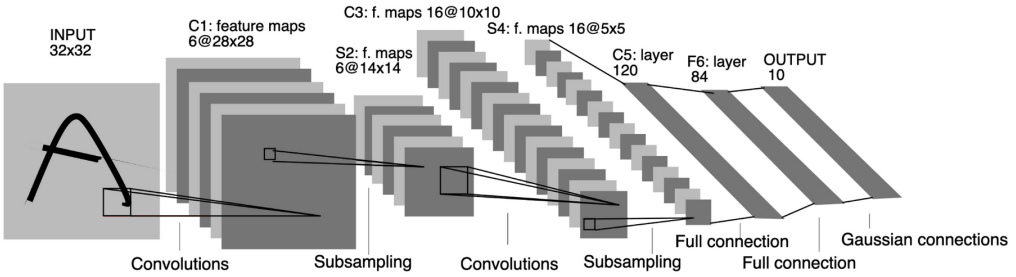


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Extração Automática de Características **Classificador convencional**

CNNs - An Example Architecture

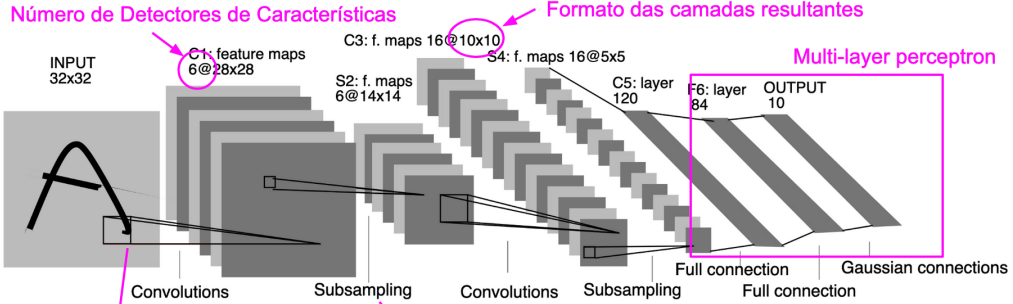


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

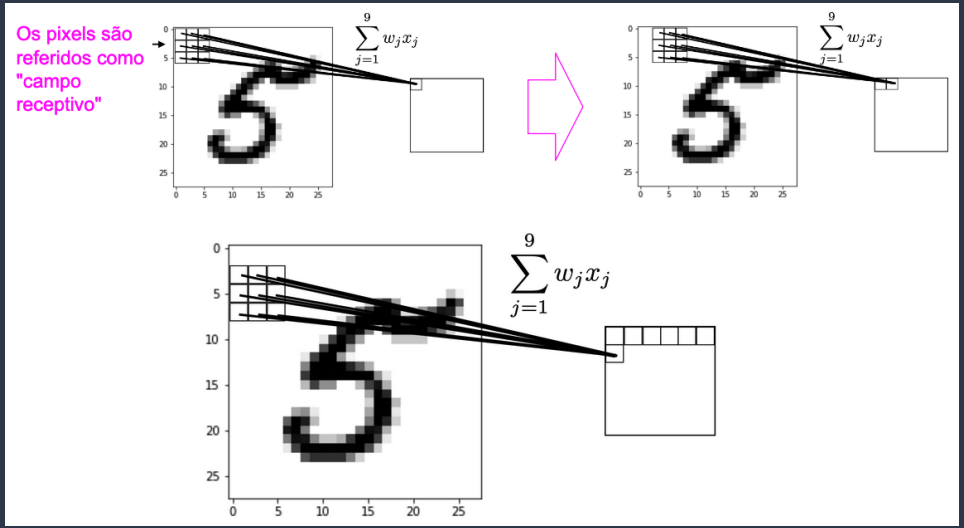
Extração Automática de Características **Classificador convencional**

"Detector de Características"
Também chamados de "kernel" ou "filtro"

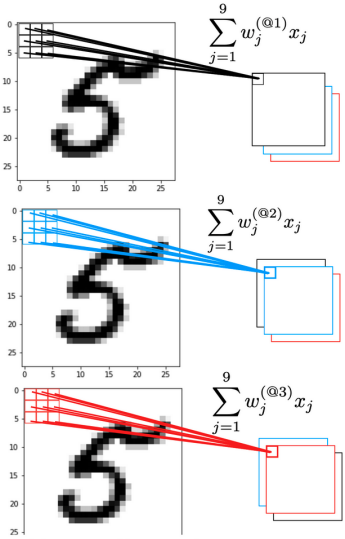
Camada conhecida como
"pooling"

Weight Sharing

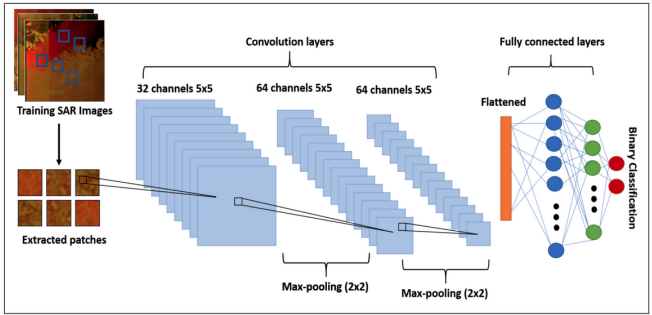
A "feature detector" (filter/*kernel*) slides over the inputs to generate a "feature map".



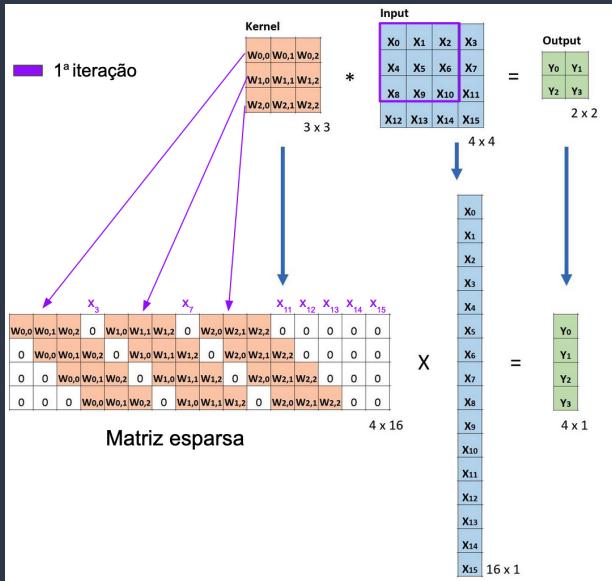
Weight Sharing



Múltiplos detectores de características podem ser utilizados para criar múltiplos mapas de características.



Sparse Connectivity



- Compartilhamento de pesos, permite a redução do número de parâmetros que serão otimizados.
- Valores zerados na matriz esparsa estão relacionados com o tamanho do kernel e do campo receptivo.

Padding and Kernels

2. Padding and Kernels

Padding

During convolution, the size of the output feature map is determined by the size of the input feature map, the kernel size, and the stride.

- Too many convolutions can impact the accuracy of the CNN if the image size is overly reduced;
- To bypass this scenario, the concept of Padding is normally used – such as adding zeros around the image;
- Padding is a process in which some pixels are added around the image before the convolution operation in order to preserve the dimensionality in the resulting image during the operation;
- This process is used because these resulting images can contain elements that make it easier for the network to identify the target class;

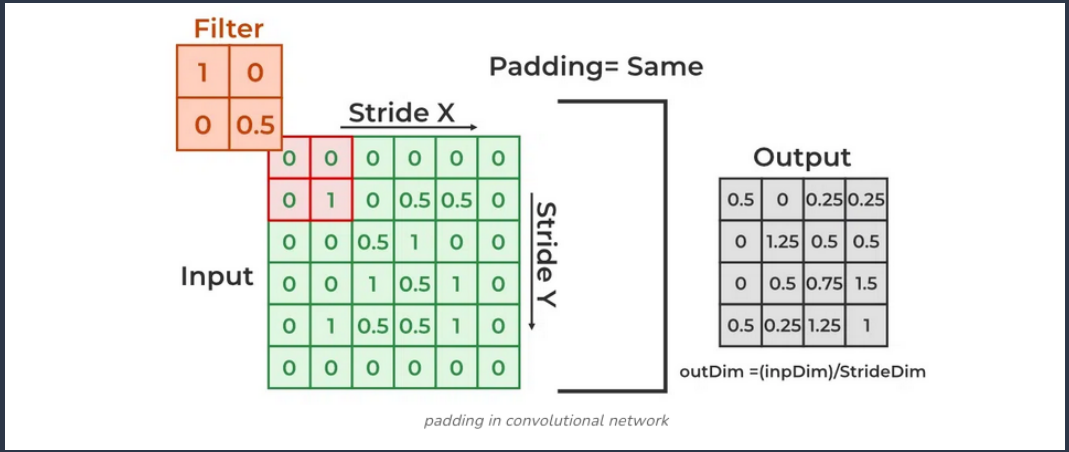
Padding

If we simply apply the kernel to the input feature map, the output feature map will be smaller than the input. This can result in a loss of information at the edges of the input feature map. To preserve edge information, we use padding.

This can be done in two ways:

- Valid padding: in valid padding, no padding is added to the input feature map, and the output feature map is smaller than the input feature map. This is useful when we want to reduce the spatial dimensions of the feature maps.
- Same padding: in same padding, padding is added to the input feature map so that the size of the output feature map is the same as the input feature map.
- This is useful when we want to preserve the spatial dimensions of the feature maps.

Padding

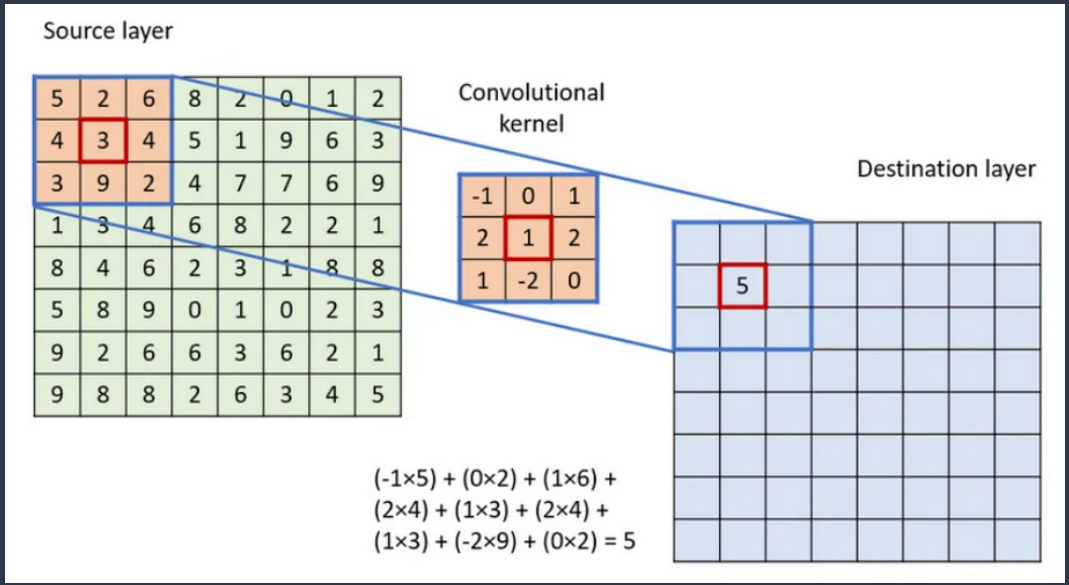

Filter
1 0
0 0.5
Padding= Same
Stride X
Stride Y
Input
0 0 0 0 0 0
0 1 0 0.5 0.5 0
0 0 0.5 1 0 0
0 0 1 0.5 1 0
0 1 0.5 0.5 1 0
0 0 0 0 0 0
Output
0.5 0 0.25 0.25
0 1.25 0.5 0.5
0 0.5 0.75 1.5
0.5 0.25 1.25 1
outDim =(inpDim)/StrideDim
padding in convolutional network

Kernels

In the context of CNNs, features refer to the significant patterns or characteristics extracted from the input data by the convolutional layers. These features represent different aspects of the input, such as edges, textures, or more complex structures.

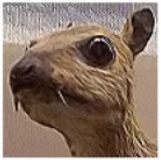
- Each convolutional kernel is responsible for detecting specific features in the input data.
- A kernel is a small 2D matrix whose content is based on the operations to be performed. A kernel maps onto the input image by simple matrix multiplication and addition; the resulting output has smaller dimensions and is therefore easier to work with.
- The shape of a kernel depends heavily on the input format of the image and the architecture of the entire network; typically, kernel sizes are $(M \times M)$, meaning a square matrix.
- The movement of a kernel is always from left to right and from top to bottom.

Kernels


Source layer
5 2 6 8 2 0 1 2
4 3 4 5 1 9 6 3
3 9 2 4 7 7 6 9
1 3 4 6 8 2 2 1
8 4 6 2 3 1 8 8
5 8 9 0 1 0 2 3
9 2 6 6 3 6 2 1
9 8 8 2 6 3 4 5
Convolutional kernel
-1 0 1
2 1 2
1 -2 0
Destination layer
5
(-1×5) + (0×2) + (1×6) +
(2×4) + (1×3) + (2×4) +
(1×3) + (-2×9) + (0×2) = 5

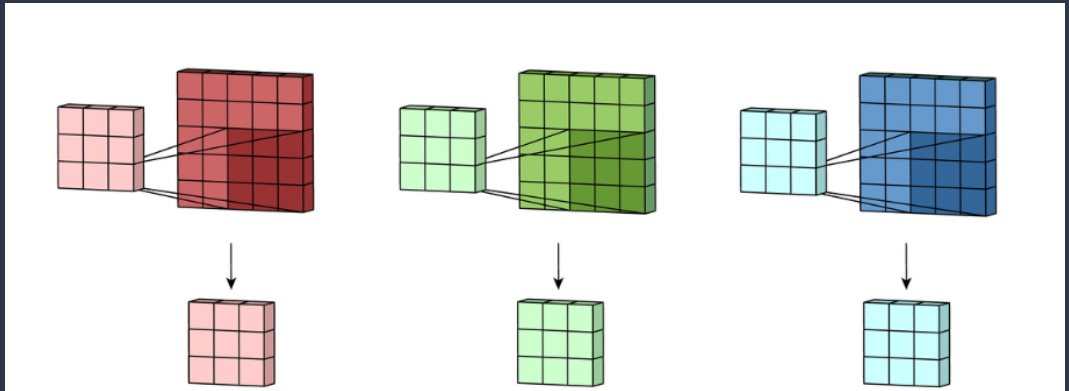
Kernels

A kernel is a small 2D matrix whose content is based on the operations to be performed. A kernel maps onto the input image by simple matrix multiplication and addition; the resulting output has smaller dimensions and is therefore easier to work with.

Original	Gaussian Blur	Sharpen	Edge Detection
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

Kernels

Kernel movement:
The stride defines the step size at which the kernel moves; for example, stride 1 makes the kernel slide one row/column at a time, and stride 2 moves the kernel by 2 rows/columns.



Pooling

3. Pooling

Pooling

Concept

Like convolutional layers, pooling operators consist of a fixed-shape window that is slid across all regions of the input according to its stride, computing a single output for each location covered by the fixed-shape window (also known as the pooling window).

- However, unlike the cross-correlation calculation of inputs and kernels in the convolutional layer, the pooling layer does not contain parameters (there is no kernel).
- Instead, pooling operators are deterministic, typically computing the maximum or average value of the elements in the pooling window.
- These operations are called max pooling (max pooling for short) and average pooling, respectively.

Pooling

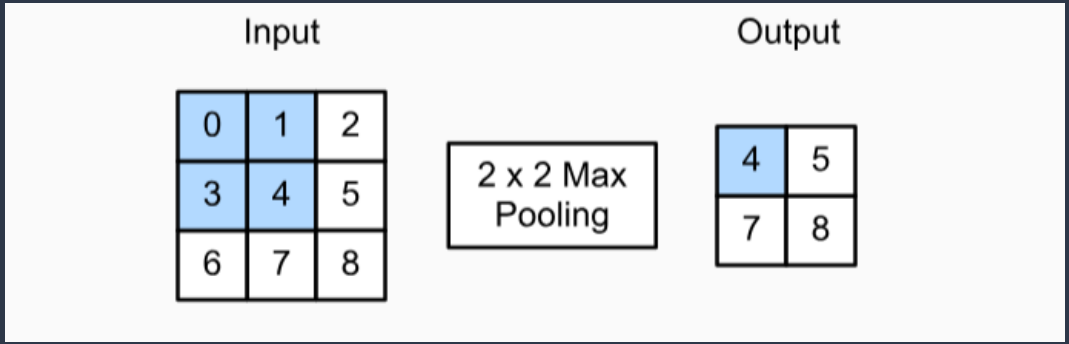


Figure 3: Max pooling with a 2×2 pooling window.

Max Pooling - Translation Invariance

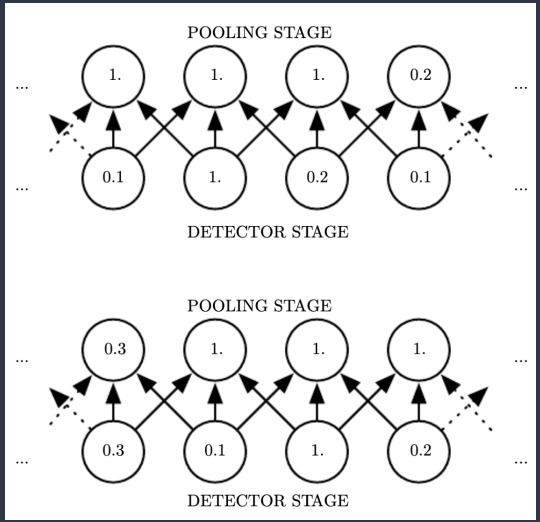


Figure 4: Effect of translation invariance with MaxPooling. A view of the same network after the inputs are shifted to the right. All input values changed (bottom), but only half of the output values were altered.

Pooling - Translation Invariance

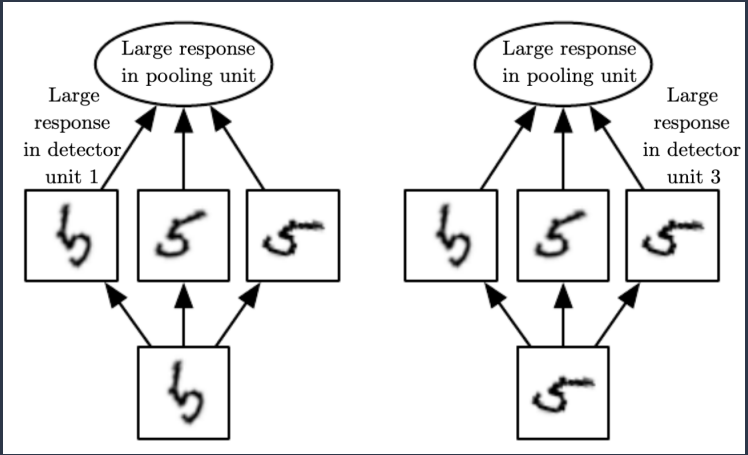
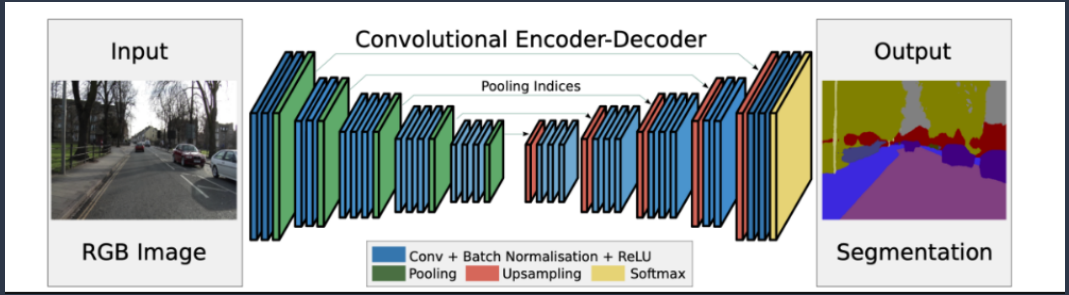


Figure 5: A pooling unit that combines several features learned with separate parameters can learn to be invariant to input transformations.

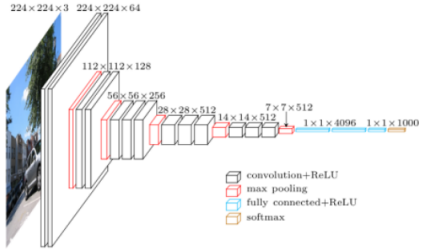
Convolutional Neural Networks: Other Applications

- Multi-label classification: final layer predicts multiple classes (e.g., image tagging);
- Object detection: finds a bounding box with an object; predicts anchor pixel and box size (e.g., faces).
- Semantic segmentation: final layer predicts an image of the original size, with each pixel belonging to a class (e.g., autonomous cars). U-net network (encoder-decoder).
- Instance segmentation: distinguishes between objects (e.g., multiple cars or faces in the image).



Convolutional Neural Networks: Other Applications

- LeNet: primeira rede convolucional bem-sucedida (60k parâmetros), desenvolvida em 1990
- AlexNet: mais profunda (60M parâmetros), vencedora do ImageNet Challenge em 2012
- GoogleNet: vencedora do ImageNet em 2014, com módulo de inception (uso de filtros em paralelo) reduzindo parâmetros (para 4M)
- VGGNet: segundo lugar em 2014, mostraram impacto da profundidade (140M parâmetros)



Deep Learning in High Energy Physics

Deep Learning in High Energy Physics

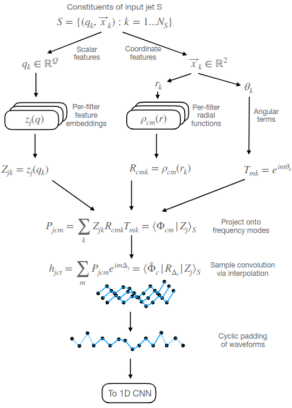
Particle Convolution for High Energy Physics

Chase Shimmin
Department of Physics
Yale University
New Haven, CT 06511
chase.shimmin@yale.edu

July 6, 2021

Abstract

We introduce the Particle Convolution Network (PCN), a new type of equivariant neural network layer suitable for many tasks in jet physics. The particle convolution layer can be viewed as an extension of Deep Sets and Energy Flow network architectures, in which the permutation-invariant operator is promoted to a group convolution. While the PCN can be implemented for various kinds of symmetries, we consider the specific case of rotation about the jet axis the $\eta - \phi$ plane. In two standard benchmark tasks, q/g tagging and top tagging, we show that the rotational PCN (rPCN) achieves performance comparable to graph networks such as ParticleNet. Moreover, we show that it is possible to implement an IRC-safe rPCN, which significantly outperforms existing IRC-safe tagging methods on both tasks. We speculate that by generalizing the PCN to include additional convolutional symmetries relevant to jet physics, it may outperform the current state-of-the-art set by graph networks, while offering a new degree of control over physically-motivated inductive biases.



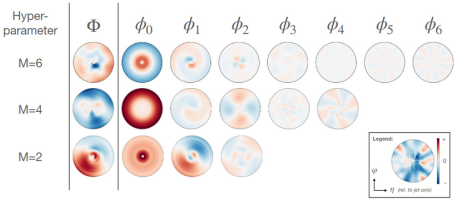


Figure 4: Visualization of the series representation of learned filter instances $\Phi(\vec{x}) = \sum_m \phi_m(r, \theta)$ for various values of the cutoff hyperparameter M.

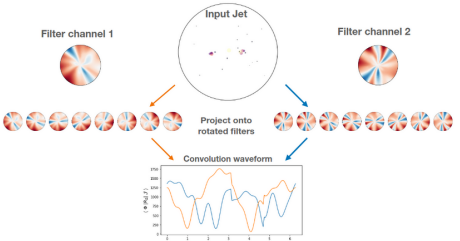


Figure 3: Illustration of the particle convolution of an example jet, represented as a point cloud, and two different filters.

Application of Deep Learning methods to analysis of Imaging Atmospheric Cherenkov Telescopes data

I. Shilon[a,*], M. Kraus[a,**], M. Büchele[a], K. Egberts[b], T. Fischer[a], T.L. Holch[c], T. Lohse[c], U. Schwanke[c], C. Steppa[b], S. Funk[a]

[a] *Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen Centre for Astroparticle Physics, Erwin-Rommel-Str. 1, D 91058 Erlangen, Germany*
[b] *Institut für Physik und Astronomie, Universität Potsdam, Karl-Liebknecht-Str. 24/25, D 14476 Potsdam, Germany*
[c] *Institut für Physik, Humboldt University of Berlin, Newtonstr. 15, D 12489 Berlin, Germany*

Deep Learning in High Energy Physics

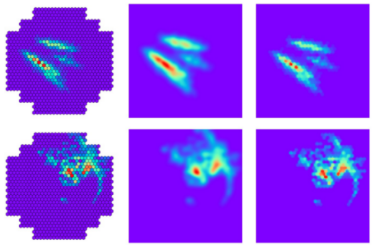
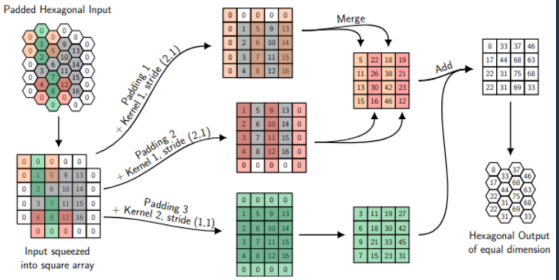
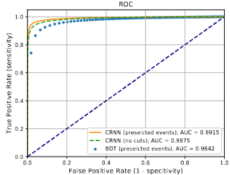
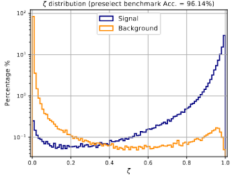


Figure 1: Telescope images of a γ-ray (*top row*) and a hadronic (*bottom row*) event. The images consist of numerous event images laid in a common camera plane, for visualization. *Left column*: Original camera pixel intensities. The square images show the same event sampled with Gaussian smoothing (*left*) and by rebinning an hexagonal histogram (*right*). The square images are oversampled with a resolution of 100 × 100.





(a) ROC curves with matching AUC values.



(b) CRNN ζ-score distribution for simulated signal and background events.

Figure 3: ROC curves for the CRNN classifier and the H.E.S.S. BDT classifier (left) and the ζ-score distribution for the CRNN classifier, obtained on the benchmark data-set with pre-selection cuts.

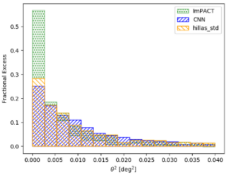


Figure 5: The squared angular distance θ^2 distribution for excess events from one flare observation of PKS 2155-304, using the Hillas, ImPACT and CNN direction reconstruction methods.

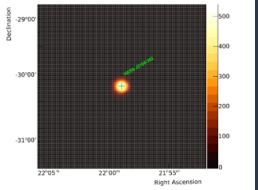


Figure 6: A two dimensional distribution of excess events observed in the direction of PKS 2155-304 for the one flare observation, using the CRNN classifier and CNN regressor.

Computational Exercise 03

- Explore Computational Exercise 03 by modifying the CNN input parameters (number of layers, number of neurons, activation functions, etc.). Evaluate the impact on the classification system's performance.

Hands-On

Exercises:

1. Choose at least 5 different learning machine architectures for the energy regression problem using data from the *Lorenzetti Showers Framework*.
2. Determine performance evaluation metrics and evaluate the different results obtained.
3. Design the models by varying configuration parameters to improve their performance.
4. Perform analyses based on batch size, epochs, number of hidden neurons/layers, etc.
5. Analyze potential statistical fluctuations in the dataset using cross-validation techniques.
6. Evaluate the use of preprocessing and input feature selection.
7. Organize the results into slides.

What is an Autoencoder?

Unsupervised Representation Learning

Up to this point, we have explored architectures requiring explicit labels or targets (like supervised energy regression). However, real-world data is often unlabeled.

The Autoencoder (AE)

An **Autoencoder** is an unsupervised neural network designed to learn a compressed, efficient representation (encoding) of input data, typically for dimensionality reduction or feature extraction.

Unsupervised Representation Learning

Up to this point, we have explored architectures requiring explicit labels or targets (like supervised energy regression). However, real-world data is often unlabeled.

The Autoencoder (AE)

An **Autoencoder** is an unsupervised neural network designed to learn a compressed, efficient representation (encoding) of input data, typically for dimensionality reduction or feature extraction.

- **Self-Supervised Nature:** It uses the input data x itself as the target label.
- **The Core Objective:** The network is trained to attempt to replicate its input to its output: $\hat{x} \approx x$.
- **The "Bottleneck":** By forcing the data through a lower-dimensional constraint, the network is restricted from simply learning the identity function, forcing it to distill essential features.

The Core Architecture

An autoencoder is structurally divided into two symmetrical components:

1. **The Encoder:** Maps the high-dimensional input data $x \in \mathbb{R}^d$ to a low-dimensional latent space vector (code) $z \in \mathbb{R}^m$ (where $m < d$).
2. **The Decoder:** Takes the latent representation z and reconstructs the original data as $\hat{x} \in \mathbb{R}^d$.

The Core Architecture

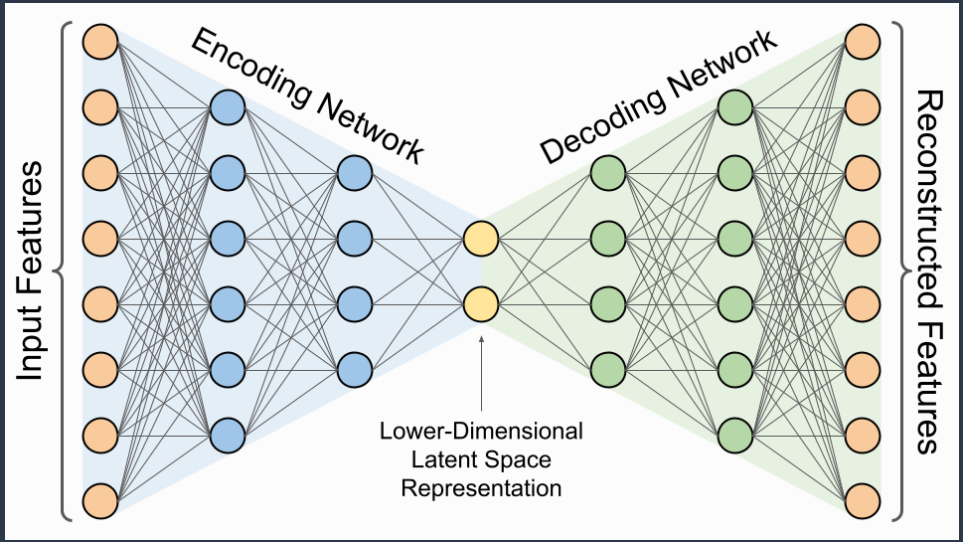
An autoencoder is structurally divided into two symmetrical components:

1. **The Encoder:** Maps the high-dimensional input data $x \in \mathbb{R}^d$ to a low-dimensional latent space vector (code) $z \in \mathbb{R}^m$ (where $m < d$).
2. **The Decoder:** Takes the latent representation z and reconstructs the original data as $\hat{x} \in \mathbb{R}^d$.

Input $(x) \longrightarrow$ **[Encoder]** $\longrightarrow$ Latent Bottleneck $(z) \longrightarrow$ **[Decoder]** $\longrightarrow$ Reconstruction $(\hat{x})$

Figure 6: Basic Operational Flow of an Autoencoder.

The Core Architecture



Mathematical Formulation

Mathematical Formulation

Let the input vector be x. The operations can be formally generalized as:

Encoder Function (f_ϕ)

$$z = f_\phi(x) = \sigma(W_e x + b_e)$$

Where W_e and b_e represent the weights and biases of the encoder, and σ is a non-linear activation function (e.g., ReLU, Sigmoid).

Mathematical Formulation

Let the input vector be x. The operations can be formally generalized as:

Encoder Function (f_ϕ)

$$z = f_\phi(x) = \sigma(W_e x + b_e)$$

Where W_e and b_e represent the weights and biases of the encoder, and σ is a non-linear activation function (e.g., ReLU, Sigmoid).

Decoder Function (g_θ)

$$\hat{x} = g_\theta(z) = \sigma'(W_d z + b_d)$$

Where W_d and b_d represent the weights and biases of the decoder, and σ' is the output activation layer.

Mathematical Formulation

Let the input vector be x. The operations can be formally generalized as:

Encoder Function (f_ϕ)

$$z = f_\phi(x) = \sigma(W_e x + b_e)$$

Where W_e and b_e represent the weights and biases of the encoder, and σ is a non-linear activation function (e.g., ReLU, Sigmoid).

Decoder Function (g_θ)

$$\hat{x} = g_\theta(z) = \sigma'(W_d z + b_d)$$

Where W_d and b_d represent the weights and biases of the decoder, and σ' is the output activation layer.

The entire network parameter space can be defined as $\Theta = \{\phi, \theta\}$.

Loss Functions The Reconstruction Error

To evaluate how well the model replicates the input data, we measure the discrepancy between x and $\hat{x}$ using a **Reconstruction Loss Function** $\mathcal{L}(x, \hat{x})$.

- **Mean Squared Error (MSE):** Used primarily when data features are continuous/real-valued.

$$\mathcal{L}_{MSE}(x, \hat{x}) = \frac{1}{2}\|x - \hat{x}\|^2$$

- **Binary Cross-Entropy (BCE):** Used if the inputs are normalized between $[0, 1]$ and treated as probabilities.

$$\mathcal{L}_{BCE}(x, \hat{x}) = -\sum_i [x_i \log \hat{x}_i + (1 - x_i) \log(1 - \hat{x}_i)]$$

The training updates parameters by minimizing the total empirical loss over the dataset:

$$\arg\min_{\phi,\theta} \sum_i \mathcal{L}(x^{(i)}, g_\theta(f_\phi(x^{(i)})))$$

Relation to PCA (Principal Component Analysis)

What happens if we remove the non-linearities from an Autoencoder?

Linear Autoencoder vs. PCA
If the encoder and decoder layers use strictly **linear activation functions**, the network parameters span the exact same subspace as **Principal Component Analysis (PCA)**.

Relation to PCA (Principal Component Analysis)

What happens if we remove the non-linearities from an Autoencoder?

Linear Autoencoder vs. PCA

If the encoder and decoder layers use strictly **linear activation functions**, the network parameters span the exact same subspace as **Principal Component Analysis (PCA)**.

Why use Deep Autoencoders instead of standard PCA?

- **Non-linear Manifolds:** Deep Autoencoders with non-linear activation functions can capture highly complex, curved multi-dimensional manifolds that linear PCA completely misses.
- **Layered Hierarchies:** Stacked layers allow the model to learn hierarchies of features (e.g., rings $\rightarrow$ shower shapes $\rightarrow$ energy structures).

Regularization Variants

The Overfitting Risk Regularization

If the latent dimension z is too large (or if the network has too many layers), the autoencoder can bypass feature extraction entirely. It could simply memorize the data or copy the input directly to the output without learning useful structures.

The Goal of Regularized Autoencoders

To constrain the network capacity using mathematical regularizers or structural noise, forcing it to prioritize the most structurally important properties of the data distribution.

We will briefly look at three standard variants:

1. Sparse Autoencoders
2. Denoising Autoencoders
3. Variational Autoencoders (VAE)

1. Sparse Autoencoders

Instead of reducing the number of hidden units in the bottleneck layer, a **Sparse Autoencoder** keeps a larger hidden layer but penalizes activation rates.

- Forces the network to only fire a small percentage of its neurons at any given time.
- Mathematically driven by adding a regularization penalty term to the loss function, such as the **Kullback-Leibler (KL) Divergence** or L_1 regularization:

$$\mathcal{L}_{Sparse} = \mathcal{L}(x, \hat{x}) + \beta \sum_{j} \mathrm{KL}(\rho \,||\, \hat{\rho}_j)$$

Where ρ is the target sparsity parameter (e.g., 0.05) and $\hat{\rho}_j$ is the average activation of hidden unit j.

2. Denoising Autoencoders (DAE)

Instead of altering the penalty optimization term, a **Denoising Autoencoder** alters the training data inputs.

$$\text{Clean } x \longrightarrow \textbf{[Add Noise]} \longrightarrow \text{Corrupted } \tilde{x} \longrightarrow \textbf{[Encoder } \rightarrow \textbf{ Decoder]} \longrightarrow \text{Clean } \hat{x}$$

- The model is fed a corrupted version $\tilde{x}$ (via Gaussian noise or dropout masking) but is trained to minimize the reconstruction loss against the **original, clean input x**.

$$\mathcal{L}_{DAE} = \mathcal{L}(x, g_\theta(f_\phi(\tilde{x})))$$

- This forces the network to learn the structural "manifold" of valid data in order to successfully undo the noise.

3. Variational Autoencoders (VAE)

Standard autoencoders are deterministic and lack structural continuity in latent space—moving between two valid codes in z might yield garbage output.

Generative Modelling with VAEs

A **Variational Autoencoder** is a probabilistic model. Instead of mapping an input to a static vector z, the encoder outputs the parameters of a probability distribution: a **Mean Vector (μ)** and a **Variance/Standard Deviation Vector (σ)**.

- Latent vectors z are sampled from $\mathcal{N}(\mu, \sigma^2)$.
- The total loss includes a **KL-Divergence term** forcing the latent distribution to look like a standard normal distribution $\mathcal{N}(0, I)$.
- Allows us to sample random vectors from $\mathcal{N}(0, I)$ and pass them through the decoder to **generate completely new synthetic data points**.

Practical Applications

Practical Applications of Autoencoders

Autoencoders are heavily used across various sectors of engineering and scientific computing:

- **Dimensionality Reduction:** Compressing high-dimensional feature systems (like a 100-ring calorimeter layout) down to critical latent spaces before passing them to secondary classification layers.
- **Anomaly / Novelty Detection:**
  - Train the Autoencoder *only* on known, nominal signatures (e.g., standard background events).
  - When presented with an anomalous or new physics signal, the model fails to reconstruct it correctly.
  - A spike in the **Reconstruction Error** flags the anomaly instantly.
- **Signal Denoising:** Filtering electronic thermal noise or cross-talk out of detector sensory readings.

Model Type	Core Trick	Primary Use Case
Linear AE	No activations	Baseline PCA emulation
Deep AE	Stacked Non-linear layers	Complex Feature Extraction
Denoising AE	Corrupted inputs $\tilde{x}$	Robust Feature Learning
Variational AE	Latent μ, σ outputs	Synthetic Data Generation

Table 1: Summary of Autoencoder Architectures.