University of Ljubljana

Faculty of Mathematics and Physics

Department of Physics

Seminar

End-to-End Classification for Discovery of New Processes in High-Energy Physics

---

Author:
Elijan Jakob Mastnak

Adviser:
prof. dr. Borut Paul Kerševan

Abstract

This work explores the use of convolutional neural networks to classify the products of high-energy particle collisions using low-level, image-based detector data. We restrict ourselves to binary classification, and, for concreteness, focus on the popular example of distinguishing Higgs boson and background events. We begin by defining end-to-end classification, explain how the data used to perform this classification is measured at the Large Hadron Collider, and outline how both fully-connected and convolutional neural networks use this data to perform classification. We conclude by presenting a concrete study involving image-based end-to-end classification, and compare end-to-end workflows to traditional classification methods based on reconstructed kinematic features.

Ljubljana, May 2021

Contents

1 Why End-to-End Classification?

1.1 What is Particle Classification?

Before discussing the merits of one classification method or the other, we must first define what we mean by classification. For our purposes, particle classification answers the following question:

> "Two high-energy particles collide. Which particles were produced as a result of their collision?"

For concreteness, we will focus on binary classification involving Higgs boson detection, in which we consider only two possible outcomes. These are:

1. a particle collision produced a Higgs boson (*signal*), or
2. a collision produced anything other than a Higgs boson (*background*).

1.2 Why End-to-End Classification?

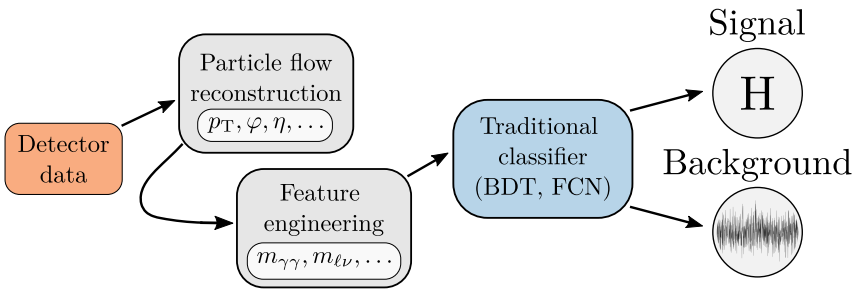

Figure 1: An abstracted traditional particle classification workflow.

Perhaps the best motivation for "end-to-end" classification comes from showing, as in Figure 1, what end-to-end classification is *not*. In particular, the steps "Particle flow reconstruction" and "Feature engineering", outlined in Section 2.4, turn out to be quite non-trivial. End-to-end classification removes these steps. In an end-to-end workflow, shown schematically in Figure 2, a classifier outputs a result computed directly from raw detector data, with minimal intermediate data processing.

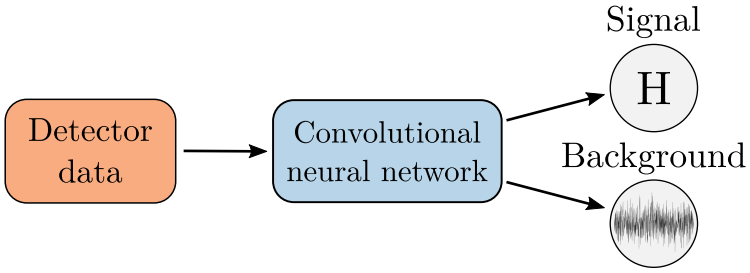

Figure 2: An end-to-end classification workflow eliminates the complicated intermediate steps involved in traditional classification (cf. Figure 1).

As implied by Figures 1 and 2, particle classification requires data measured by a particle detector. In the next section, we explore the nature of this detector data, and outline how the data is produced and measured at the Large Hadron Collider.

2 How the LHC Produces and Measures Collision Data

For our purposes, the Large Hadron Collider (LHC) is a synchrotron accelerator that collides protons at teravolt-order center-of-mass energy [12]. Each collision produces a cascade of secondary particles, whose energies and trajectories are measured in specialized detectors like CMS and ATLAS. The purpose of this section is to explain:

1. the process leading up to a proton collision at the LHC,
2. the physical principles behind a detector's measurement instruments, and
3. the image-like format of low-level detector data used by end-to-end classifiers.

2.1 The CERN Accelerator Chain

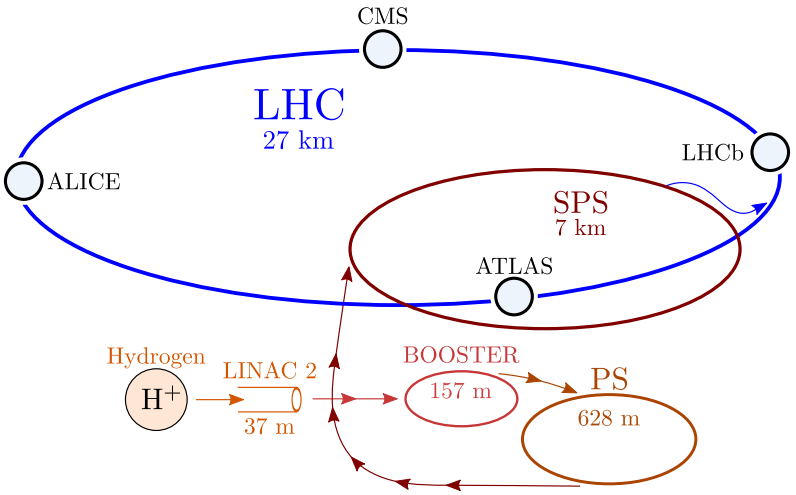

Figure 3: The components of the CERN accelerator complex relevant to this work. Adapted from [15].

The story begins with a bottle of hydrogen gas. A strong electric field strips the hydrogen atoms of their electrons, leaving behind only protons. These protons are accelerated through the linear accelerator *Linac2*,[1] setting in motion a journey through the maze of synchrotron boosting stages shown in Figure 3 and culminating at the LHC. Depending on operating conditions, the LHC accelerates protons to an ultimate kinetic energy of up to 6.5 TeV; these protons travel around the LHC in two opposing beamlines in bunches of roughly 10^{11} particles, separated in time by 25 ns. The beamlines cross, allowing for proton-proton collisions, at four nominal collision points, around which are centered, in anticipation, the LHC's four main detectors: ALICE, CMS, LHCb, and ATLAS.

2.2 Collision and Detection

Most protons in opposing beams zoom past each other without appreciable interaction. Rarely, however, two protons collide head-on, setting in motion a chain of interactions

[1] In the LHC's 2020 high luminosity upgrade, Linac2 is replaced with Linac4, which accelerates negative hydrogen ions instead [19].

which ultimately produces a cascade of familiar elementary particles such as electrons, protons, neutrons, muons, photons, and neutrinos. These secondary particles are collectively called the collision's *decay signature*, and fly outward in all directions from the primary collision point through a surrounding particle detector. This brings us to a key concept:

> The exotic particles of interest in high-energy physics (e.g. a Higgs boson) decay far, far before they can interact with the detector. The detector measures only the secondary particles further down the decay chain.

In other words, *we have no way of detecting a Higgs boson directly. The best we can do is infer its presence from the nature of the collision's decay signature.* Thus, to progress, we need a quantitative description of a decay signature.

2.3 Quantifying a Decay Signature

Fundamentally, a particle detector measures the following decay signature properties:

1. the trajectory of particles through the detector, and
2. the energy deposited in the detector by detected particles.

Trajectories are measured with instruments called *trackers*, while energy is measured with *calorimeters*. From trajectory and energy, and by noting which particles appear in which subdetector, we can reconstruct a number of secondary quantities, including particle momentum, identity, and the positions of production and decay vertices. In this section we first define the detector coordinate system, then discuss each measurement instrument in turn. For concreteness, we will focus on the instruments used at the Compact Muon Solenoid (CMS) detector [8], but similar concepts apply to most particle detectors currently used in high-energy physics. For better orientation, each of the four CMS sub-detectors discussed in this section appear, in correct scale, in the diagram of the CMS detector shown in Figure 5 two pages below.

2.3.1 The Detector Coordinate System at the CMS

The CMS coordinate system is best understood visually, as shown in Figure 4. The detector uses a hybrid Cartesian/cylindrical coordinate system, in which the origin coincides with the nominal collision point, while the x, y, and z axes point towards the center of the LHC, vertically upward, and along the beamline, respectively. The azimuthal angle φ is measured in the xy plane, while the polar angle θ is measured in the yz plane. Conventionally, the polar angle is given in terms of a quantity called *pseudorapidity*,[2] denoted by η and defined as

$$\eta \equiv -\ln\left(\tan\frac{\theta}{2}\right) \implies \theta = 2\arctan e^{-\eta}. \tag{1}$$

[2]Particle physicists work with pseudorapidity, and a related quantity called rapidity, because differences in these quantities are Lorentz-invariant to boosts along the beam axis. For our purposes, it is enough to view pseudorapidity as just an alternate representation of the polar angle θ.

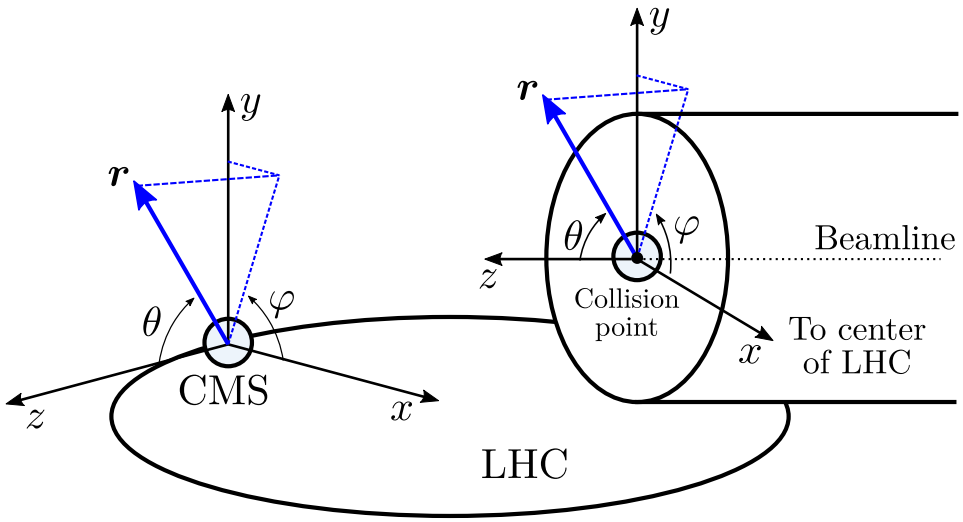

Figure 4: Two perspectives of the coordinate system used at the CMS detector.

2.3.2 Trackers

Trackers measure the trajectory (position with respect to time) of particles flying out from the collision point. The basic tracker building blocks are microscopic silicon pixels and strips with dimensions of order 10 µm to 100 µm, each connected to its own electronic read-out channel. Each pixel (strip) has an intrinsic electric field, similar in principle to the built-in electric field in a pn junction's depletion region.

The CMS tracker [10] consists of millions of these silicon elements arranged in concentric, cylindrical layers around the nominal collision point. An incident particle passing through a silicon pixel (strip) excites the atoms in the pixel's (strip's) depletion region, creating electron-hole pairs. The silicon element's electric field then accelerates the electrons and holes to opposite pixel (strip) faces, leading to a measurable charge pulse called a *hit*. Hits from successive layers are combined to reconstruct an incident particle's trajectory.

Importantly, the CMS tracker is immersed in a 4 T magnetic field, generated by the CMS's namesake *solenoidal* superconducting magnet. Because of this magnetic field, charged particles in the Tracker bend under the influence of the Lorentz force, which allows physicists to reconstruct the momenta of charged particles from the curvatures of their Tracker trajectories.

2.3.3 Electromagnetic Calorimeters

Electromagnetic calorimeters (ECALs) measure the energy of particles that interact with matter via the electromagnetic interaction. At the CMS, the basic ECAL building block is a lead tungstate ($PbWO_4$) scintillator crystal connected to a photodetector [7].

The basic working principle is as follows: high-energy incident particles interact with the $PbWO_4$ scintillators (electrons via bremsstrahlung, high-energy photons predominantly via pair production) to produce secondary *electromagnetic showers*—cascades of lower-energy photons, electrons, and positrons. Shower particles excite the $PbWO_4$ scintillators, which emit blue-light *scintillation photons* during the relaxation process. These scintillation photons reach the crystal's attached photodetector

(avalanche photodiodes in the ECAL barrel and vacuum phototriodes in the ECAL end caps [7]), where they free electrons via the photoelectric effect. After further amplification, these photoelectrons produce a measurable electric current whose amplitude encodes the energy deposited by the initial incident particle.

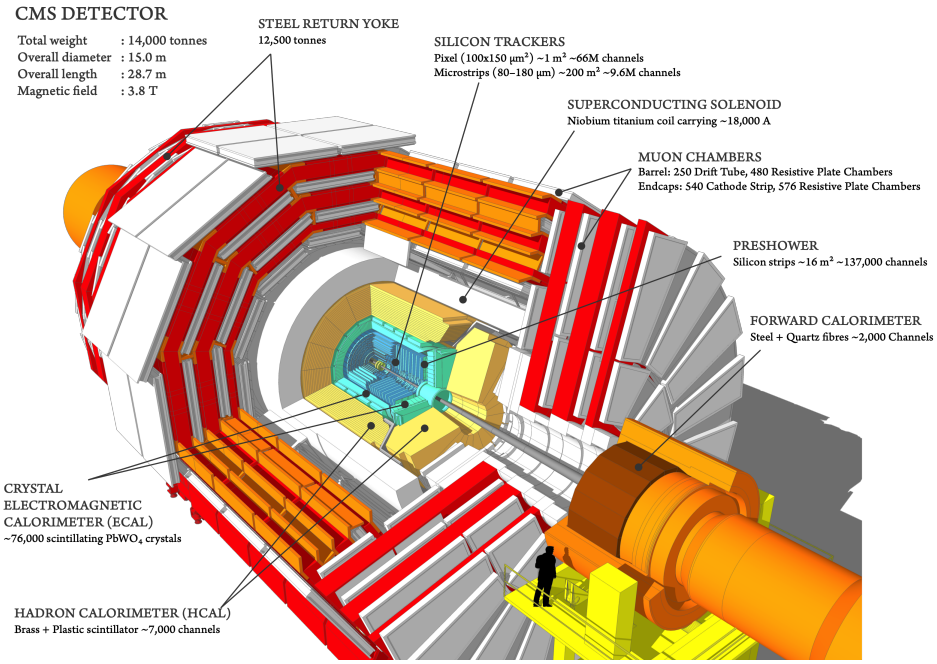

Figure 5: A cutaway diagram of the CMS detector, showing each of the sub-detectors (silicon trackers, ECAL, HCAL, and muon detectors) discussed in this work. [17]

2.3.4 Hadronic Calorimeters

Hadronic calorimeters (HCALs) measure the energy of particles that interact with matter via the strong interaction. The CMS HCAL consists of alternating layers of absorbers—brass and steel plates—and scintillation detection modules—plastic scintillator tiles connected to hybrid photodiodes [9].

A high-energy particle interacts with the absorber material, primarily via the strong interaction, to produce a *hadronic shower*—a cascade of lower-energy hadronic particles. The shower particles excite the plastic scintillator tiles, which emit blue-light scintillation photons; these photons are then converted to a measurable electric current via the photoelectric effect by the tiles' attached photodiodes. Just like in the ECAL, the current's amplitude encodes the energy deposited by an incident particle, while the position of the activated scintillator tile encodes the particle's position.

2.3.5 Muon Detection

For the purposes of this work, this section may be skipped without loss of continuity. However, for completeness, we now briefly outline the CMS's namesake muon detection system. The CMS muon detector is a gas-based detector, whose basic module is a rectangular *drift tube* enclosing a mixture of argon and CO_2 gas (10 to 20 percent CO_2 by volume). Through the chamber's center runs a thin steel anode wire (radius 50 µm),

held at a kilovolt-order potential difference relative to copper-coated electrodes glued to the chamber's outer surface [16].

A high-energy muon incident on the drift tube ionizes the internal gas, freeing electrons and positive ions. These ions are accelerated across the tube's potential difference to the anode wire and cathode electrodes, respectively, resulting in a measurable pulse of electric charge. The number of freed ions is proportional to the energy deposited in the drift tube by the incident muon, so the signal amplitude, like in ECAL and HCAL, encodes the muon's energy; the location of the signal pulse along the electrodes reveals the muon's position.

Three other muon detector components, namely resistive plate chambers, cathode strip chambers, and gas electron multipliers, rely on the same physical principle of gas ionization and ion collection. For more information, interested readers are referred to Refs. [3] or [16] for a friendly or technical discussion, respectively.

Importantly, the muon detection system resides at the very outer layer of the CMS. Since muons are the only familiar decay product (besides neutrinos, which do not ionize gas) to pass through the ECAL and HCAL unimpeded, we can be reasonably sure any particle registered in the muon detector is indeed a muon.

2.4 Low-Level Detector Data and How It Is Used

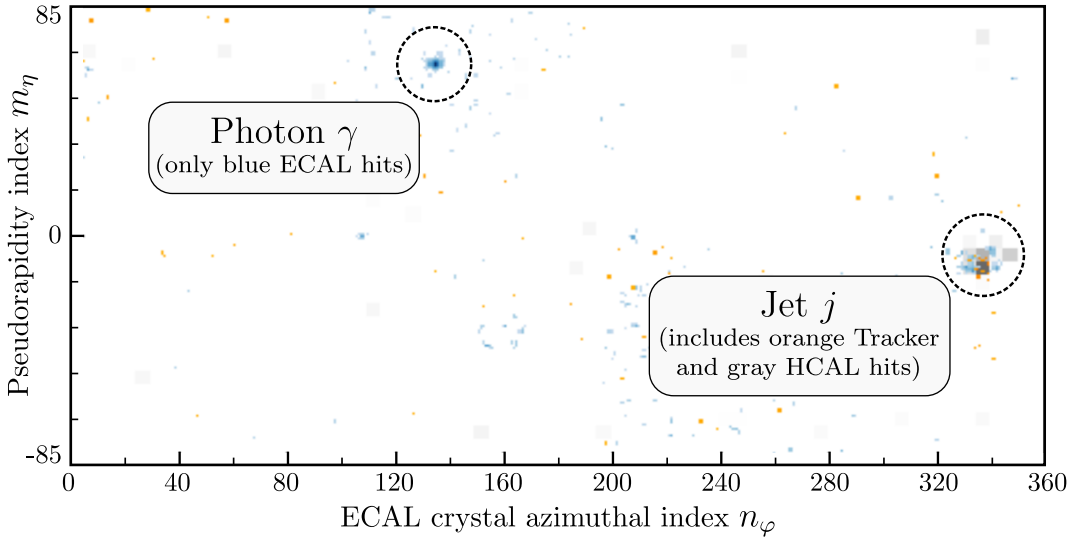

Figure 6: A composite image of a photon+jet (background) event at the CMS. Tracks appear in orange log scale, ECAL hits in blue log scale, and HCAL hits in gray linear scale. Pixels correspond to physical $PbWO_4$ scintillator crystals in the ECAL barrel's 360×170 crystal grid, which spans $\varphi \in [0, 2\pi)$ and $|\eta| < 1.479$. Adapted from [1].

If we combine the Tracker hits and ECAL/HCAL signal pulses and project these quantities onto the detector's $\varphi\eta$ plane, we get something like the image shown in Figure 6. This image, from a simulated photon+jet event at the CMS [5], shows the raw Tracker (position) and ECAL/HCAL (energy) information encoding the event, projected onto the ECAL barrel's 360×170 grid of $PbWO_4$ scintillator crystals.

Figure 6 serves as our basic model of low-level detector data: an image-like grid with two spatial dimensions (φ, η) and three detector channels, in which pixel intensity

corresponds to either (i) charge deposited in the Tracker, or (ii) energy deposited in ECAL or HCAL. In each case, the position of a given pixel in the image has a well-defined physical correspondence to the position of a measurement module (e.g. Tracker pixel, ECAL scintillation crystal) in the detector. Equipped with this form of image-based data, we have three options:

(a) *End-to-end classification*: use the low-level data as is, and feed this data into a convolutional neural network, which outputs a predicted classification result.

(b) *Kinematic-based classification*, in which we feed the raw data into a *particle flow reconstruction* algorithm,[3] which reconstructs *kinematic features*, for example transverse momentum (relative to the beam axis) and pseudorapidity (Eq. 1), describing the decay products. We then use this kinematic data as input into a fully-connected neural network (FCN), which outputs a classification result.

(c) Reconstruct kinematic features from the low-level data as above, then use these kinematic features to hand-engineer *high-level features*, manually designed to separate signal from background events on an experiment-by-experiment basis (for example, decay particle invariant masses at which we expect high signal production). Then, feed these high-level features, often in combination with the kinematic features described above, into a FCN or BDT classifier. See Ref. [2] for a study comparing kinematic-based and high-level classification.

Schematically, classification workflows (b) and (c) resemble Figure 7.

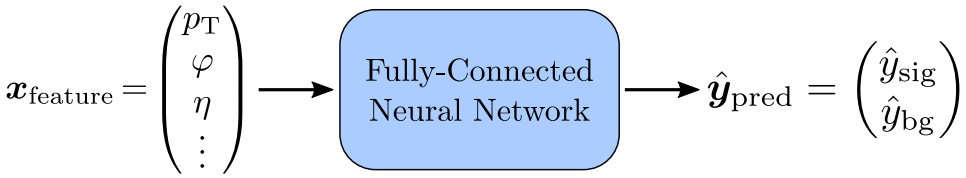

Figure 7: The binary classification process using a FCN. The outputs $\hat{y}_{\text{sig}}$ and $\hat{y}_{\text{bg}}$ represent the predicted probabilities that the inputted feature vector $\boldsymbol{x}_{\text{feature}}$ corresponds to either signal or background, respectively, as explained in Section 3.1.

3 Fully-Connected Networks

In this section we take a look inside the "Fully-Connected Neural Network" box in Figure 7. Suppose, as in Figure 7, that we have a set of kinematic properties encoding a collision event (found, for example, with the particle flow reconstruction algorithm mentioned in Section 2.4), which we pack into a vector $\boldsymbol{x}$. In machine learning terminology [13], each individual kinematic property is called a *feature*, and the two possible outcomes (signal or background) are called *classes*. The correct class (either signal or background) for a given event is called the event's *target* or *label*, and the complete set of features describing a single event, together with the label,

[3] Particle flow reconstruction falls largely beyond the scope of this work. As an example, we can reconstruct a particle's transverse momentum from the curvature of its trajectory in the detector's magnetic field, but we will otherwise treat reconstruction as a "black box" that outputs kinematic quantities describing a collision. Interested readers are referred to Ref. [18].

is called an *instance*. Finally, the complete set of available instances (the complete set of collision events) forms a *dataset*. With the vocabulary lesson out of the way, we are equipped to formally describe a neural network. But first, we should reveal supervised machine learning's dirty secret:

> The dataset used to train a neural network is entirely *simulated*.[4]

Why? Well, we *know* the outcome of each simulated collision event. Thus, our dataset comes with correct labels for every single instance. Using this large dataset of correctly-labeled data, we can train a neural network to give accurate results even when, as in practical applications, the correct results aren't known beforehand.

3.1 Understanding a Binary Classifier's Output

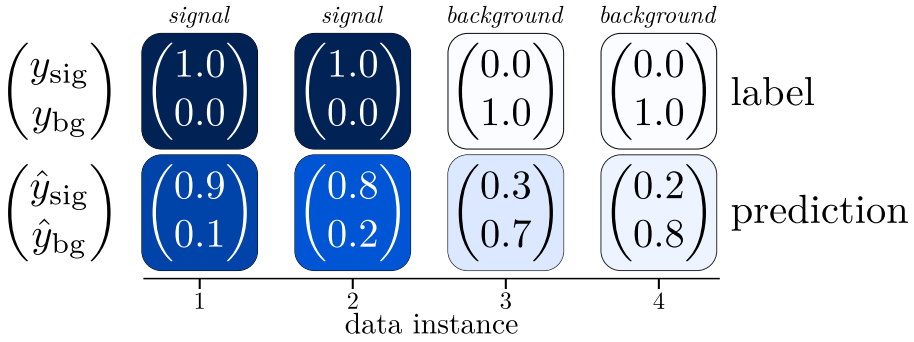

Figure 8: A representative binary classification output for four hypothetical data instances. While labels $\boldsymbol{y}$ give perfect "one" or "zero" answers, a classifier's actual predictions $\hat{\boldsymbol{y}}$ represent a probability for each class in the continuous range $[0, 1]$.

We now pause to clarify what a binary classifier's output looks like and how to interpret the result. First, we decide on an order in which to quote the possible classes: in this work we will quote signal first, then background. The label, represented by a vector $\boldsymbol{y}$, equals one at the position of the correct class and zero at the position of the incorrect class. In the signal-first convention, a generic label vector reads $\boldsymbol{y} = (y_{\text{sig}}, y_{\text{bg}})$; the label $\boldsymbol{y} = (1, 0)$ represents signal while $\boldsymbol{y} = (0, 1)$ represents background. Meanwhile, a classifier's prediction, denoted by $\hat{\boldsymbol{y}} = (\hat{y}_{\text{sig}}, \hat{y}_{\text{bg}})$, represents the predicted *probabilities* that an event corresponds to either signal or background; these two values should sum to one. Figure 8 summarizes these ideas visually.

3.2 An Overview of a Fully-Connected Network

For better orientation, Figure 9 shows the architecture of a fully-connected neural network. The basic building of a FCN is a *neuron*—represented by the circular nodes in Figure 9. Neurons are then grouped into one-dimensional stacks to form *layers*. A FCN consists of a single *input layer*, an arbitrary number of *hidden layers*, and a single *output layer*, which outputs the classification scores discussed in Section 3.1.

[4]Simulated collision data is constructed with sophisticated algorithms involving Monte-Carlo sampling, a theoretical physical framework such as the Standard Model, and a geometrical model of the particle detector. Like particle flow reconstruction, this falls beyond the scope of this work.

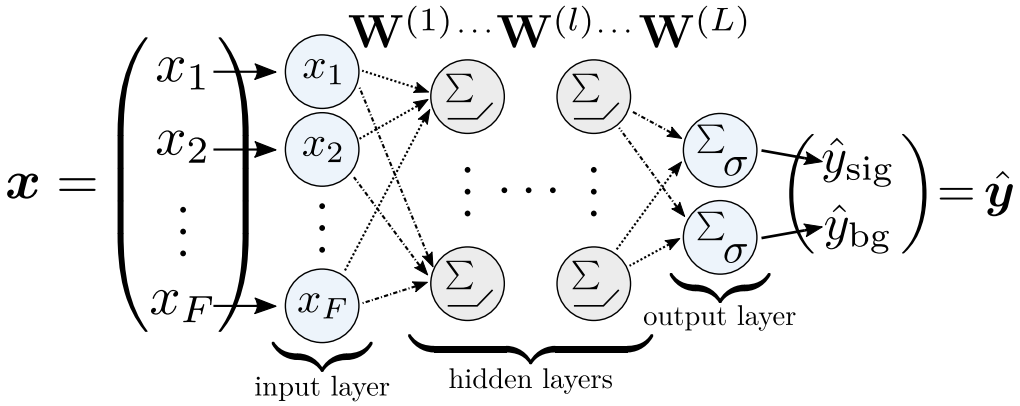

Figure 9: A fully-connected network's internal architecture; each neuron beyond the input layer connects to *every* neuron in the previous layer. The figure uses the notation from Appendix A. The $\sum$ (sum) symbol represents the multiply-add neuron operation in Equation 2, while the hinge and σ symbols represent the ReLU and softmax activation functions, shown in Figure 10 and Equation 3, respectively.

In this section we build up to a big-picture interpretation of FCN-based classification as the optimization of a vector-valued function, a concept which should be familiar to physicists. That said, before proceeding we first refer interested readers to Appendix A, which thoroughly explains the nuts and bolts of a FCN's internal architecture in the language of the machine-learning literature. However, while it might help demystify the "black box" nature of neural networks, this formal treatment can be skipped without loss of continuity.

3.2.1 The Basic Architecture of a Fully-Connected Network

A neuron may be viewed as a multi-variable scalar function; in analogy with neuroscience, a neuron's output is called its *activation*, which we will denote by a. When considering the forward flow of information through a neural network, we must consider the following two points. Each hidden layer neuron, which we will also refer to as the "current neuron",

(a) receives as input the activations $a_1^{(\text{prev})}, a_2^{(\text{prev})}, \ldots, a_{n_{\text{prev}}}^{(\text{prev})}$ of all n_{prev} neurons in the previous layer, and

(b) passes its scalar output a to all neurons in the next layer.

As required by point (a) and visualized in Figure 9, each hidden-layer neuron has input connections to *every* neuron in the previous layer, hence the name "fully-connected" network. The strength of each of these connections is parameterized by a unique scalar *weight* $w_j \in \mathbb{R}$, where $j = 1, 2, \ldots, n_{\text{prev}}$; in other words, w_j parameterizes the current neuron's connection to the j-th neuron in the previous layer. For compactness, we write the current neuron's weights as the vector

$$\boldsymbol{w} = \left(w_1, w_2, \ldots, w_{n_{\text{prev}}}\right) \in \mathbb{R}^{n_{\text{prev}}};$$

similarly, the previous layer's activation values are written as the vector

$$\boldsymbol{a}_{\text{prev}} = \left(a_1^{(\text{prev})}, a_2^{(\text{prev})}, \ldots, a_{n_{\text{prev}}}^{(\text{prev})}\right) \in \mathbb{R}^{n_{\text{prev}}}.$$

In addition to its n_{prev} weights, the current neuron is parameterized by a single scalar bias b, which serves as an additive constant which can translate the neuron's activation value.

Next, to understand the current neuron's output, we proceed in two steps:

1. First, the input activation values $a_j^{(\text{prev})}$ from the previous layer's neurons are multiplied with each of their corresponding weights w_j, summed, and translated by the current neuron's bias b to produce a pre-activation value z given by

$$z = w_1 \cdot a_1^{\text{prev}} + \cdots + w_n \cdot a_n^{\text{prev}} + b = \boldsymbol{w} \cdot \boldsymbol{a}_{\text{prev}} + b \in \mathbb{R}. \tag{2}$$

   Note that this is just the dot product of the current neuron's weight vector with the previous layer's activation vector, plus the current neuron's bias, i.e. a linear function of the input activations.

2. Second, the current neuron's pre-activation value z is passed through a *non-linear activation function* $f_{\text{a}} : \mathbb{R} \to \mathbb{R}$ to produce the scalar activation value

$$a = f_{\text{a}}(z) = f_{\text{a}}\big(\boldsymbol{w} \cdot \boldsymbol{a}_{\text{prev}} + b\big) \in \mathbb{R}.$$

Crucially, using *non-linear* activation functions allows the network to form non-linear decision boundaries in the high-dimensional feature space of the inputted feature vectors; these non-linear boundaries enormously improve classification potential. In practice, deep networks predominantly use the rectified linear unit (ReLU) function (Figure 10) or one of its variants;[5] I will use the notation f_{a} for generality.

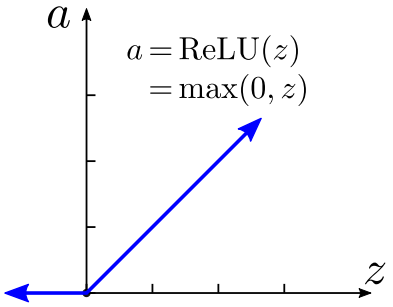

Figure 10: The ReLU function

The output layer's activation function is different. At least in classification problems, it is chosen so that the sum of the output layer's activations equals one, giving the interpretation of the output activations as a probability distribution over the possible classes. A common output layer activation function, applicable to an arbitrary number of, say, C classes, is the softmax function

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, C \text{ and } \boldsymbol{z} \in \mathbb{R}^C. \tag{3}$$

Moving one step up the network hierarchy from a single neuron to an entire layer (which we'll call the "current layer"), we pack the weight vectors of each of the n neurons in the current layer column-wise into a weight matrix $\mathbf{W}$ given by

$$\mathbf{W} = \begin{pmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \cdots & \boldsymbol{w}_n \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_{\text{prev}},1} & w_{n_{\text{prev}},2} & \cdots & w_{n_{\text{prev}},n} \end{pmatrix} \in \mathbb{R}^{n_{\text{prev}},n},$$

where the weight $w_{j,i}$ parameterizes the connection between the i-th neuron in the current layer and the j-th neuron in the previous layer. Similarly, we pack the biases of each neuron in the current layer into a single bias vector

$$\boldsymbol{b} = \big(b_1, b_2, \ldots, b_n\big)^\top \in \mathbb{R}^n,$$

[5] See Wikipedia's list of common activation functions for a comprehensive list.

where b_i denotes the bias of the i-th neuron in the current layer. Using this matrix formalism, the entire current layer's pre-activation output is compactly written

$$\boldsymbol{z} = \mathbf{W}^\top \boldsymbol{a}_{\text{prev}} + \boldsymbol{b} \in \mathbb{R}^n,$$

while the current layer's activation values read

$$\boldsymbol{a} = f_{\text{a}}(\boldsymbol{z}) = f_{\text{a}}\Big(\mathbf{W}^\top \boldsymbol{a}_{\text{prev}} + \boldsymbol{b}\Big) \in \mathbb{R}^n,$$

where the activation function f_{a} is understood to act on the vector $\boldsymbol{z}$ element-by-element. The current layer's activation values $\boldsymbol{a}$ then serve as input to the next layer, and the cycle repeats. In this way, a feature vector describing a particle collision event propagates through the entire network from input layer to output layer, as shown in Figure 9. The eventual activation values in the output layer serve as the classification scores discussed in Section 3.1, from which we can read off the inputted collision's predicted class.

3.2.2 FCN Classification as a High-Dimensional Optimization Problem

We now describe the "big-picture" interpretation promised in this section's introduction. A single neuron is a multi-variable scalar-valued function, parameterized by a weight vector $\boldsymbol{w}$ and bias b, which takes as input the previous layer's activation $\boldsymbol{a}_{\text{prev}}$ and outputs the scalar activation value

$$a_i = f_{\text{a}}(\boldsymbol{w} \cdot \boldsymbol{a}_{\text{prev}} + b) \in \mathbb{R}.$$

Letting F and C denote the number of features and classes, respectively, the entire network is a multi-variable vector-valued function $\boldsymbol{h} : \mathbb{R}^F \to \mathbb{R}^C$, parameterized by the L weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\boldsymbol{b}^{(l)}$ of each layer. The network takes as input a feature vector $\boldsymbol{x} \in \mathbb{R}^F$ and produces as output a vector of predicted classification scores $\hat{\boldsymbol{y}} \in \mathbb{R}^C$, where the component $\hat{y}_c$ gives the probability that the feature vector $\boldsymbol{x}$ corresponds to the c-th class, $c \in \{1, 2, \ldots, C\}$. In this light:

> A FCN classification problem is the optimization problem of finding the optimal values $\mathbf{W}^{(l)}_{\text{opt}}$ and $\boldsymbol{b}^{(l)}_{\text{opt}}$ of each layer's weights and biases such that the network's prediction $\hat{\boldsymbol{y}}$ for the class of an inputted feature vector $\boldsymbol{x}$ gives the best possible approximation to the true result encoded by the label vector $\boldsymbol{y}$.

3.2.3 A Brief Look at Optimization

Machine learning optimization works by making a neural network's predictions "less bad" (rather than, say, improving a positive metric). To proceed, we need a quantitative metric encoding how "incorrect" a network's prediction $\hat{\boldsymbol{y}}$ is, relative to a known label vector $\boldsymbol{y}$. This metric is called *loss*. For a classifier with C classes, loss is given by a scalar-valued *loss function* $L : \mathbb{R}^C \to \mathbb{R}$, which takes as input a predicted classification score $\hat{\boldsymbol{y}}$ and label vector $\boldsymbol{y}$, and returns a scalar value L encoding the difference between $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$. For concreteness, an example loss function commonly used for classification problems is the *categorical cross entropy* function

$$L(\hat{\boldsymbol{y}}; \boldsymbol{y}) = -\sum_{c=1}^{C} y_c \ln \hat{y}_c. \tag{4}$$

For this function, recalling the prediction and label vector structure from Section 3.1 in which $y_c \in \{0, 1\}$ and $\hat{y}_c \in [0, 1]$, we see that $L = 0$ if $\hat{\boldsymbol{y}} = \boldsymbol{y}$, while L grows increasingly larger as the elements in $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$ increasingly differ.

Put briefly, we optimize the network's weights and biases by minimizing the loss function using standard numerical methods for multi-dimensional minimization problems, but adapted to very large parameter spaces and huge datasets. Interested readers are referred to Appendix A.3 and Chapter 6.5 of Ref. [14].

4 Convolutional Neural Networks

An end-to-end classification workflow using convolutional neural networks appears in Figure 11. Our goal in this section, just like in Section 3 for FCNs, is to outline what occurs inside the "Convolutional Neural Network" box in Figure 11. As for FCNs, we describe the key ideas here and relegate some technical details to Appendix B, which may be skipped without loss of continuity.

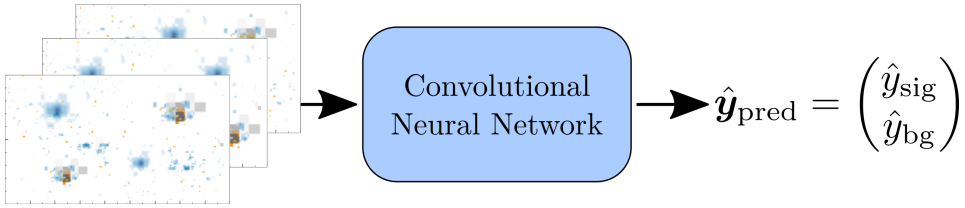

Figure 11: The binary classification process using a CNN. Instead of using reconstructed kinematic quantities, the classifier works directly with image-based detector data (cf. Figure 7).

Suppose, as shown schematically in Figure 11, that we have a set of image-based, low-level detector data of the same form as Figure 6; these images will serve as input to our CNN. Let's begin by analyzing the properties of our input data. Paraphrasing from the excellent explanation in Ref. [11], our image-based detector data:

(a) is stored as multi-dimensional arrays,

(b) has one axis—the detector channel—used to access different views of a collision event from the three CMS subdetectors (Tracker, ECAL, and HCAL), and

(c) has two spatial axes—for the coordinates φ and η—with well-defined spatial structure in which ordering matters.

Point (c) is especially important. In an image such as Figure 6, the relative positions and intensities of the pixels encode a wealth of relevant physical information about particle trajectory and energy, as discussed in Sections 2.3 and 2.4. Even if you forget everything else about CNNs, you should remember this:

> Convolutional neural networks are designed to *preserve and leverage the information encoded in an input image's spatial structure* (in a way that FCNs, which are limited to one-dimensional vector inputs, cannot).

We thus need a novel, space-preserving way for convolutional networks to interact with their input images. Instead of flattening the input into a large vector, we might

imagine "scanning" the two-dimensional image with a small, also two-dimensional "filter", which moves across the image and builds up a map of distinguishing features, such as bright spots, curves, or edges. This is the essence of a *discrete convolution*, the core (and namesake) operation of CNNs and the subject of the next section.

4.1 Discrete Convolution

A discrete convolution involves two multidimensional objects—an input image and a *kernel* or *filter*. The kernel is a pixel-like grid like the input image, but always smaller in height and width. Each kernel pixel is assigned a scalar *weight*, and the kernel as a whole a single scalar *bias*; these kernel weights and biases are a CNN's tunable parameters, and replace the neuron weights and biases in a FCN [13, 14].

During discrete convolution,[6] the kernel slides across the input image from left to right and top to bottom, and at each possible position the kernel and input elements are multiplied element-wise and summed to produce a single scalar value. The set of these scalar products at each possible kernel position within the input image produces the output, called an *output feature map*, again an image-like pixel grid. This is awkward to explain in words but straightforward visually—hence Figure 12. Importantly, a discrete convolution's output has the same dimensional form as its input, and can thus be used as the *input feature map* to another convolutional layer.

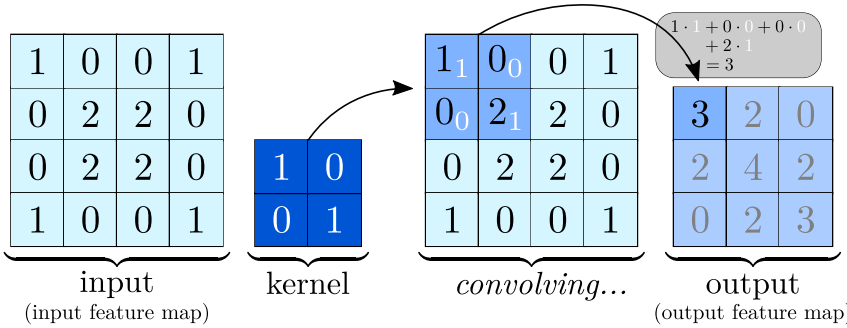

Figure 12: A toy example of a discrete convolution. The kernel is placed at each possible position in the input, and the kernel and input are multiplied element-wise and summed to form the output values. The kernel's bias is omitted for conciseness.

In practice, images are three-dimensional—they contain multiple channels, such as the three subdetector channels in our CMS detector data or the three color channels in RGB images. In this case the convolutional kernel is also three dimensional, with a separate kernel channel for each image channel, as shown in Figure 13. After convolution, a kernel's outputs are summed across the channel axis to produce a scalar value for each spatial position of the kernel within the input image. The end result is again a two-dimensional output feature map, as in Figure 13. Mathematical relationships between kernel, input, and output size are given in Appendix B.3

In practice, as shown schematically in Figure 15, CNNs also convolve a single input feature map with *multiple* kernels—just like FCNs employ multiple neurons in each fully-connected layer. Each kernel captures different feature information (edges, curves, contrasting colors...) describing the input image, so using more kernels

[6] Formally, the operation commonly called discrete convolution is actually cross-correlation; true convolution employs a kernel whose elements are reflected across the spatial axes. This technicality is irrelevant in practice; see Chapter 9.1 of Ref. [14] for a more thorough discussion.

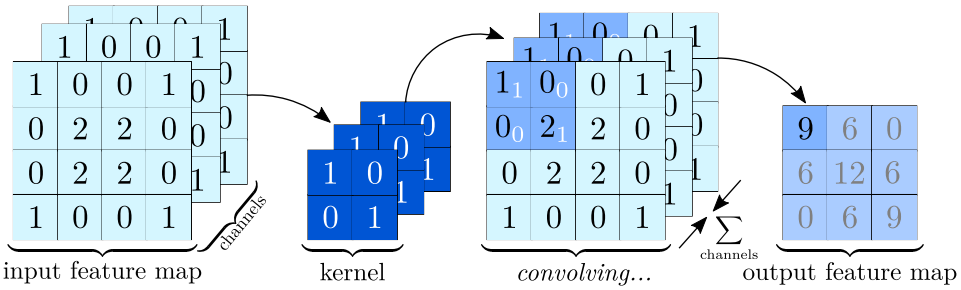

Figure 13: Discrete convolution with multiple channels in the input; cf. Figure 12. The convolutions from each channel are summed across the channel axis to produce a single-channel output. In practice, of course, the input and kernel would have different values in each channel. The kernels' biases are omitted for conciseness.

improves potential classification power. In this case, a convolutional layer's output is three dimensional (with as many channels as the number of kernels used for convolution) and serves as the multi-channel input to a subsequent convolutional layer.

4.2 Pooling

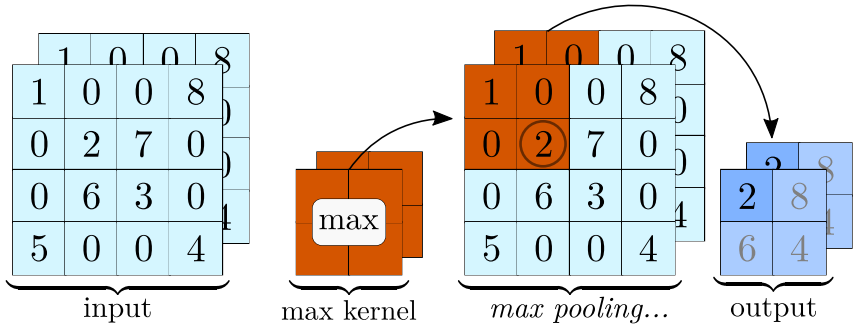

Figure 14: The max pooling operation. The kernel moves in non-overlapping patches across the input and outputs the maximum pixel value at each patch. Note that, unlike discrete convolution (Figure 13) pooling preserves the channel dimension.

Convolutional networks also employ a second, simpler operation called *pooling*. Pooling serves two purposes; these are to:

- make the CNN's output invariant to local translations of the input image, and
- spatially downsample an input without introducing new learnable parameters.

Pooling involves a pooling kernel, which is analogous to the convolutional kernel, but simpler. Most commonly, the pooling kernel slides across an input feature map's spatial dimensions in non-overlapping patches, and outputs the maximum pixel value at each kernel position. Pooling preserves channel dimensions, so a pooling stage's output always has the same number of channels as the input, as shown in Figure 14.

In principle, the pooling kernel could also move across overlapping or arbitrarily-strided patches, and output, say, the average pixel value (*average pooling*) instead of the maximum (*max pooling*). However, max pooling with non-overlapping strides appears most commonly in modern CNNs [13].

4.3 The Basic Architecture of a Convolutional Network

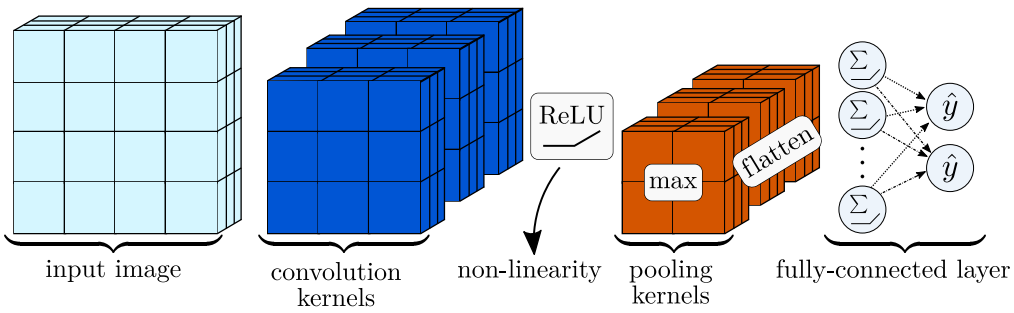

Figure 15: The important stages of a convolutional network in their typical sequence.

Figure 15 shows the typical sequence of operations in a CNN: an input image is processed by multiple kernels in a convolutional layer, passed element-wise through a non-linearity (most commonly a ReLU), and downsampled in a pooling layer.[7] The process then repeats, with one stage's output feature map used as the input to the next stage. Finally, the output of the last pooling layer is flattened into a one-dimensional vector and passed through a single fully-connected layer to produce classification scores in an identical format as already seen for FCNs in Section 3.1.

For CNNs, we tune the convolutional kernels' weights and biases instead of the neuron weights and biases in FCNs. The general principles of loss, optimization and backpropagation through computational graphs discussed in Appendix A.3 in the context of FCNs still apply to CNNs, and we will not repeat them here.

5 End-to-End Classification in Practice

5.1 A Case Study in End-to-End Classification

For concreteness, we now summarize an actual study involving end-to-end classification: the work of Andrews et al. in Ref. [1]. This study uses convolutional neural networks to distinguish diphoton Higgs boson decays from two similar background events using simulated CMS data. The study considers the following three processes: (i) gluon fusion Higgs production to diphoton decay $gg \to H^0 \to \gamma\gamma$ (signal); (ii) quark-antiquark annihilation to diphoton decay $q\bar{q} \to \gamma\gamma$ (background); and quark-antiquark annihilation to photon+jet decay $q\bar{q} \to \gamma j$ (background). Representative Feynman diagrams for the signal and diphoton background process appear in Figure 16, while the relevant datasets may be found in Refs. [4], [5], and [6].

The two background processes are chosen specifically to represent two common challenges in high-energy physics particle classification. These are:

(a) *irreducible backgrounds*—the diphoton background has identical decay products to the signal process, as shown in Figure 16—and

(b) *unresolved decay products*—for reasons irrelevant to this work,[8] the jet in the

[7] In practice, a convolutional stage's outputs are usually also normalized to zero mean and unit variance in a *normalization layer*, which is empirically found to improve training performance [13].

[8] For the sake of completeness, the jet is electromagnetically enriched to deposit its energy primarily in the ECAL via a neutral meson decaying to two merged photons [1, 5].

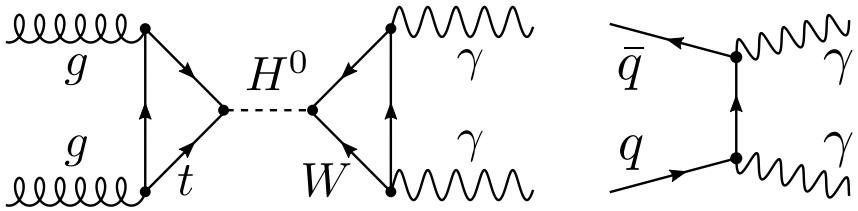

Figure 16: Representative Feynman diagrams showing the signal (left) and diphoton background (right) processes in Andrews' study. For the purposes of this work, it suffices to note that both processes share identical $\gamma\gamma$ backgrounds.

> γj background appears in the ECAL detector as a single, photon-like cluster, so the γj decay signature appears similar to its $\gamma\gamma$ counterpart.

The study uses a 15-layer variant of a common CNN architecture called a *residual network* (ResNet), which employs a structure called a "residual block", in which inputted feature maps are allowed to bypass convolutional layers; interested readers are referred to Chapter 14, page 457 of Ref. [13]. As input, the CNN classifier accepts the same image-like data shown in Figure 6, and outputs *three* classification scores $\hat{y}_{\text{sig}}$, $\hat{y}_{\text{bg}_{\gamma\gamma}}$, and $\hat{y}_{\text{bg}_{\gamma j}}$, representing the predicted probabilities for each of the three processes considered in the study. For reference, the study also classifies the same datasets using a kinematics-based FCN classifier, discussed in Section 3. This FCN is trained on the transverse momenta, pseudorapidities, and azimuthal angles of the decay particles, and the results are compared to those obtained by the CNN.

5.1.1 Discussion of Results

We now briefly examine one of the study's more interesting results. Namely, we compare the performance of the FCN and the two CNN classifiers when distinguishing $H \to \gamma\gamma$ signal and $q\bar{q} \to \gamma j$ background events; one CNN is trained only with ECAL data (**CNN, ECAL**) and one with Tracker, ECAL and HCAL data (**CNN, all**). The classifications results are shown in Figure 17 with a receiver-operating characteristic (ROC) curve, which shows a classifier's background rejection as a function of classification efficiency. From Figure 17, we immediately see that, in this particular case, the CNN classifiers substantially outperform their FCN counterpart.

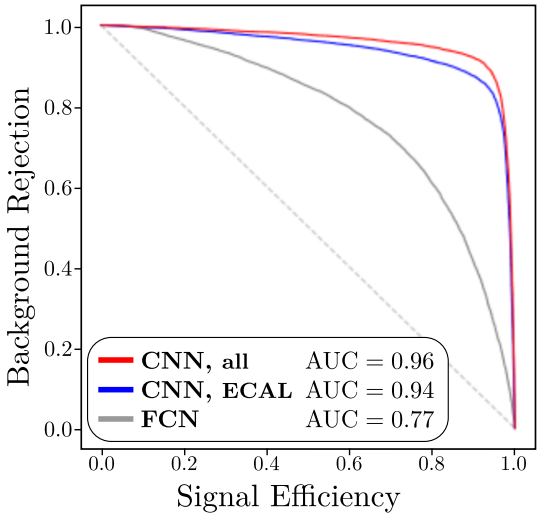

Figure 17: An ROC curve showing the results for a kinematic-based FCN (gray) and two end-to-end CNN (blue and red) classifiers for distinguishing $H \to \gamma\gamma$ signal and γj background. Adapted from [1].

To interpret this discrepancy, it helps to remember what the relevant detector data actually looks like; recall that a representative γj background event appears in Figure 6. A human, equipped with

the power of vision, can immediately distinguish the photon and jet in Figure 6. Aside from the obvious fact that the hadronic jet carries Tracker and HCAL hits, we also notice subtle differences in shower distribution—the photon's hits are concentrated and circular, while the jet's are more dispersed and dragged out elliptically along the φ axis. However, we might struggle to give an answer if the only information we received were the momentum and (φ, η) coordinates of the two events. But that is precisely the information seen by a kinematic-based FCN! A CNN, meanwhile, is designed to preserve and learn from the entire image's spatial structure; hence the improved performance seen in Figure 17. In other words, CNN classifiers using low-level detector data show promise for distinguishing between processes with subtle differences in shower distribution, and thus offer a powerful classification tool for high-energy physicists in the search for new physical processes.

5.2 Concluding Thoughts

Using raw detector data, without further processing, preserves the maximum available physical information about a particle collision event. Since convolutional networks are better suited to learning from the spatial information encoded in raw detector data than fully-connected networks, CNNs and the end-to-end classifiers employing them show a clear advantage over kinematic feature-based classifiers for processes distinguished by subtle difference in particle shower distributions.

The benefits of end-to-end classification are numerous. Aside from their greatest benefit—preserving and learning from spatial information—end-to-end workflows provide a general, widely applicable framework to particle classification. Because of their low-level inputs, end-to-end classifiers mitigate our reliance on particle flow reconstruction of kinematic features, and potentially eliminate completely the need for manual engineering of high-level features on a case-by-case basis.

Additionally, although not discussed in this work, using traditional FCN classifiers with processes involving many (or a variable number of) decay products poses a considerable technical challenge [1]. For CNNs such issues are avoided altogether—one simply inputs raw detector data and training labels, and the CNN develops its classification power from the spatial distribution of the particle showers. As such, CNN-based end-to-end classifiers offer a promising tool for analyzing the increasingly complex processes and weaker signals involved in modern particle physics and the search for physics beyond the Standard Model.

A A More Technical Discussion of FCNs

We begin by introducing the notation[9] needed to describe a FCN:

- Let F denote the number of features and C the number of classes. A binary classifier with two classes has $C = 2$, but we use C for generality.
- Let L denote the number of hidden layers (so that the network has $L + 1$ layers when including the input layer).
- Let $l = 0, 1, 2, ..., L$ index the layers, with $l = 0$ the input layer and $l = L$ the output layer.
- Let n_l be the number of neurons in layer l, and let i, where $i = 1, \ldots, n_l$, index the neurons in each layer. A neuron's position in the network is uniquely defined by the layer l and index i.

A.1 Weights and Biases

The purpose of the weights and biases appears immediately below in Appendix A.2.

Consider an arbitrary i-th neuron in the network's l-th layer, and, for better semantics, let $n_{\text{prev}} \equiv n_{l-1}$ denote the number of neurons in the previous layer. This generic l-th layer neuron is assigned:

1. A total of n_{prev} scalar *weights* $w_{j,i}^{(l)} \in \mathbb{R}$, contained in a weight vector

$$\boldsymbol{w}_i^{(l)} = \left(w_{1,i}^{(l)}, w_{2,i}^{(l)}, \ldots, w_{n_{\text{prev}},i}^{(l)}\right)^\top \in \mathbb{R}^{n_{\text{prev}}}. \tag{5}$$

   Each l-th layer neuron connects to all n_{prev} neurons in the previous layer, and the n_{prev} weights parameterize these connections (see Figure 9). Going forward, I will drop the $^{(l)}$ superscript from individual weights for easier reading.

2. A single scalar bias $b_i^{(l)} \in \mathbb{R}$.

Moving up in the hierarchy, the full l-th layer, which contains n_l neurons, is assigned

1. A single weight matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{n_{\text{prev}}, n_l}$ given by

$$\mathbf{W}^{(l)} = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n_l} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n_l} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_{\text{prev}},1} & w_{n_{\text{prev}},2} & \cdots & w_{n_{\text{prev}},n_l} \end{pmatrix} \equiv \left(\boldsymbol{w}_1^{(l)}\, \boldsymbol{w}_2^{(l)} \cdots \boldsymbol{w}_{n_l}^{(l)}\right) \in \mathbb{R}^{n_{\text{prev}}, n_l}. \tag{6}$$

   The weight matrix $\mathbf{W}^{(l)}$ has one column for each neuron in the l-th layer; this column holds the corresponding neuron's weight vector from Equation 5.

2. A single bias vector $\boldsymbol{b}^{(l)}$ given by

$$\boldsymbol{b}^{(l)} = \left(b_1^{(l)}, b_2^{(l)}, \ldots, b_{n_l}^{(l)}\right)^\top \in \mathbb{R}^{n_l}, \tag{7}$$

   which holds each of the biases of the n_l neurons in the l-th layer.

[9] Note that some of the notation used in this work is not standard, but chosen on pedagogical grounds for better semantics (e.g. l to index layers, c to index classes, a for activation value, etc...).

A.2 Response and Activation

A.2.1 Pre-Activation Response

As shown in Figure 9, the input layer passes the feature vector $\boldsymbol{x} \in \mathbb{R}^F$ to each neuron in the first hidden layer. The *pre-activation response* $z_i^{(1)} \in \mathbb{R}$ of the i-th neuron in the first layer is a scalar value given by the weighted sum

$$z_i^{(1)} = \boldsymbol{w}_i^{(1)} \cdot \boldsymbol{x} + b_i^{(1)} = w_{1,i}^{(1)} \cdot x_1 + w_{2,i}^{(1)} \cdot x_2 + \cdots w_{F,i}^{(1)} \cdot x_F + b_i^{(1)},$$

where $\boldsymbol{w}_i^{(1)} \in \mathbb{R}^F$ and $b_i^{(1)}$ are the weight vector (Equation 5) and bias of the i-th neuron in the first layer. Moving up in the hierarchy, the pre-activation values of all n_1 neurons in the first layer may be compactly written as a vector $\boldsymbol{z}^{(1)} \in \mathbb{R}^{n_1}$ given by the matrix equation

$$\boldsymbol{z}^{(1)} = \big(\mathbf{W}^{(1)}\big)^\top \boldsymbol{x} + \boldsymbol{b}^{(1)},$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{n_0,n_1} \equiv \mathbb{R}^{F,n_1}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^{n_1}$ are the first layer's weight matrix and bias vector, respectively (introduced in Equations 6 and 7).

A.2.2 A Neuron's Activation

The *activation value* $a_i^{(1)}$ of the i-th neuron in the first layer is the scalar value

$$a_i^{(1)} = f_{\mathrm{a}}^{(1)}(z_i^{(1)}) \in \mathbb{R},$$

where $f_{\mathrm{a}}^{(1)}$ is the first layer's *activation function*. The activation values of all n_1 neurons in the first layer may be compactly written as a vector $\boldsymbol{a}^{(1)} \in \mathbb{R}^{n_1}$ given by the matrix equation

$$\boldsymbol{a}^{(1)} = f_{\mathrm{a}}(\boldsymbol{z}^{(1)}) = f_{\mathrm{a}}\Big(\big(\mathbf{W}^{(1)}\big)^\top \boldsymbol{x} + \boldsymbol{b}^{(1)}\Big),$$

where the activation function f_{a} is understood to act on the vector $\boldsymbol{z}^{(1)}$ element-by-element. In other words, $f_{\mathrm{a}}(\boldsymbol{z}^{(1)})$ is shorthand for

$$f_{\mathrm{a}}(\boldsymbol{z}^{(1)}) \equiv \begin{pmatrix} f_{\mathrm{a}}(z_1^{(1)}) \\ f_{\mathrm{a}}(z_2^{(1)}) \\ \vdots \\ f_{\mathrm{a}}(z_{n_1}^{(1)}) \end{pmatrix} \in \mathbb{R}^{n_1}.$$

In practice, all hidden layers use the same activation function, so I will omit the superscript $^{(l)}$ for conciseness.

Moving forward through the network, the first hidden layer's activation values $\boldsymbol{a}^{(1)} \in \mathbb{R}^{n_1}$ are passed forward to each of the n_2 neurons in the second hidden layer. The pre-activation values of the second hidden layer's neurons are then

$$\boldsymbol{z}^{(2)} = \big(\mathbf{W}^{(2)}\big)^\top \boldsymbol{a}^{(1)} + \boldsymbol{b}^{(2)} \in \mathbb{R}^{n_2},$$

while the activation values of the neurons in the second hidden layer are

$$\boldsymbol{a}^{(2)} = f_{\mathrm{a}}(\boldsymbol{z}^{(2)}) = f_{\mathrm{a}}\Big(\big(\mathbf{W}^{(2)}\big)^\top \boldsymbol{a}^{(1)} + \boldsymbol{b}^{(2)}\Big) \in \mathbb{R}^{n_2}.$$

More generally, the activation values of the n_l neurons in the l-th hidden layer are

$$\boldsymbol{a}^{(l)} = f_{\mathrm{a}}(\boldsymbol{z}^{(l)}) = f_{\mathrm{a}}\Big(\big(\mathbf{W}^{(l)}\big)^\top \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\Big) \in \mathbb{R}^{n_l}.$$

A.3 Optimization

A.3.1 Loss

To review from the brief discussion in Section 3.2.3, for a classifier with C classes, loss is given by a scalar-valued *loss function* $L : \mathbb{R}^C \to \mathbb{R}$, which takes as input a predicted classification score $\hat{\boldsymbol{y}}$ and label vector $\boldsymbol{y}$, and returns a scalar value L encoding the difference between $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$. In practice, for computational efficiency during optimization, loss is computed on the predictions from a *mini-batch* of $M \sim 128$ feature vectors instead of individual instances. In this case, the loss generalizes to

$$\mathcal{L} = \frac{1}{M} \sum_{m=1}^{M} L_m(\hat{\boldsymbol{y}}_m; \boldsymbol{y}_m), \tag{8}$$

which is just the average of the losses associated with each instance in the mini-batch.

A.3.2 The Optimization Process

Equipped with a metric for a network's performance (i.e. loss), we can formulate the machine learning problem as finding the weights and biases minimizing the network's loss $\mathcal{L}$. The general spirit of the optimization process follows below:

1. Input into the network a mini-batch $\left\{(\boldsymbol{x}_m, \boldsymbol{y}_m)\right\}_{m=1}^{M}$ of $M \sim 128$ instances; the network outputs a corresponding batch of predictions $\left\{\hat{\boldsymbol{y}}_m\right\}_{m=1}^{M}$.
2. Compute the loss L_m associated with each individual prediction $\hat{\boldsymbol{y}}_m$ in the mini-batch using, say, Equation 4, then compute the entire mini-batch loss $\mathcal{L}$ using Equation 8.
3. Compute the gradients $\nabla_{\mathbf{W}}^{(l)}(\mathcal{L})$ and $\nabla_{\boldsymbol{b}}^{(l)}(\mathcal{L})$ of the mini-batch loss $\mathcal{L}$ with respect to the weights $\mathbf{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ in each of the network's layers.
4. Adjust the values of each layer's weights $\mathbf{W}^{(l)}$ and biases $\boldsymbol{b}^{(l)}$ using the just-computed gradients of the mini-batch loss according to

$$\mathbf{W}_{\text{new}}^{(l)} \leftarrow \mathbf{W}_{\text{old}}^{(l)} - \eta \nabla_{\mathbf{W}}^{(l)}(\mathcal{L}) \qquad \text{and} \qquad \boldsymbol{b}_{\text{new}}^{(l)} \leftarrow \boldsymbol{b}_{\text{old}}^{(l)} - \eta \nabla_{\boldsymbol{b}}^{(l)}(\mathcal{L}),$$

   where η is a training parameter called the *learning* rate, with typical values in the range 10^{-6} to 10^{-1}, often adjusted dynamically during the training process [13]. Since the gradients point in the direction of increasing $\mathcal{L}$, adding *negative* gradient moves the network in the direction of decreasing loss.
5. Repeat steps 1 to 4 with a new mini-batch. In principle, the network's loss will decrease with increasing iterations, and the network's predictions $\hat{\boldsymbol{y}}$ will begin to approach the true target values $\hat{\boldsymbol{y}}$. Assuming all goes well,[10] stop iterating when performance plateaus, or when satisfied with the network's results.

The magic occurs in step 4. However, I have pulled step 3—computing the loss's gradient with respect to the network's weights and biases—out of thin air. In practice, these gradients are computed with an algorithm called *backpropagation*, outlined in the next section.

A.3.3 Backpropagation

At its core, backpropagation is simply repeated application of the chain rule for differentiation. In preparation, we first inventory the network's complete set of operations (additions, multiplications, activation functions...), beginning with the input layer and ending with the loss $\mathcal{L}$. We then organize these operations into a structure called a *computational graph*, with each operation assigned a node in the graph. Backpropagation consists of two steps:

[10] I'm sweeping some details under the rug here. Among other things, a real-world training process would include regularization, a validation metric to monitor over-training, and mechanism for early stopping. These fall beyond our scope, and interested readers are referred to Ref. [13].

1. First, in a procedure called a *forward pass*, we propagate an inputted feature vector through the network's computational graph from input to output (*upstream*) until arriving at a scalar loss $\mathcal{L}$, as described in Sections A.2 and A.3.1.
2. Second, in a procedure called a *backward pass*, we begin with the just-computed loss and work backwards towards the inputs (*downstream*), calculating the local gradient of each node in the computational graph with respect to the node's immediate inputs. We then use the chain rule to propagate these gradients further downstream through the graph, eventually arriving back at the input layer and computing the gradient of loss with respect to each layer's weights and biases in the process. These gradients are then used to adjust the network's parameters, as in step 4 in the previous section. The process then repeats with a new feature vector.

For completeness, we now review the chain rule, beginning with the simple scalar case. Using our established FCN notation, we consider loss $\mathcal{L} = f_{\mathcal{L}}(a)$ as a function of an output neuron's activation $a = f_{\text{a}}(z)$, which is in turn a function of the neuron's pre-activation value z. In this case, the derivative of $\mathcal{L}$ with respect to z is

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}. \tag{9}$$

Figure 18 shows a toy example of backpropagation, using Equation 9, in a single-neuron network. Although the network is trivial, the figure still captures the essence of backpropagation: repeated application of the chain rule to propagate gradients downstream through a computational graph to reach the derivative of loss with respect to weights and biases.

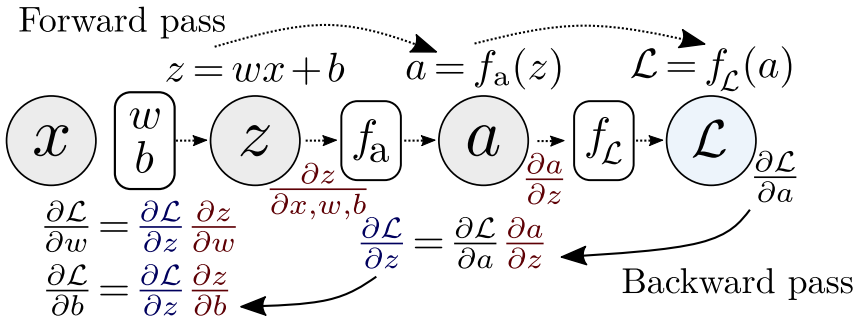

Figure 18: The principles of backpropagation in a single-neuron toy network.

For realistic cases we need the chain rule's vector generalization. Suppose $\mathcal{L} \in \mathbb{R}$ is a function of $\boldsymbol{a} \in \mathbb{R}^n$, which is in turn a function of $\boldsymbol{z} \in \mathbb{R}^m$; we write this as

$$\mathcal{L} = \boldsymbol{f}_{\mathcal{L}}(\boldsymbol{a}) = \boldsymbol{f}_{\mathcal{L}}(\boldsymbol{f}_{\text{a}}(\boldsymbol{z})),$$

where $\boldsymbol{f}_{\mathcal{L}} : \mathbb{R}^n \to \mathbb{R}$ and $\boldsymbol{f}_{\text{a}} : \mathbb{R}^m \to \mathbb{R}^n$. Then the gradient of $\mathcal{L}$ with respect to $\boldsymbol{z}$ is

$$\nabla_{\boldsymbol{z}} \mathcal{L} = \left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}}\right)^{\top} \cdot \nabla_{\boldsymbol{a}} \mathcal{L} \in \mathbb{R}^m,$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \in \mathbb{R}^{n \times m}$ is the function $\boldsymbol{f}_{\text{a}}$'s Jacobian matrix. In practice, one would use an even more general tensor formulation applicable to objects with an arbitrary number of indices. For scalar loss $\mathcal{L}$ as a function of a tensor[11] A, in turn a function of a tensor Z, this reads

$$\nabla_{\mathsf{Z}} \mathcal{L} = \sum_j \left(\nabla_{\mathsf{Z}} \mathsf{A}_j\right) \frac{\partial \mathcal{L}}{\partial \mathsf{A}_j},$$

where j is a tuple index that runs over all possible indices into the tensor A. Further discussion falls beyond our scope; interested readers are referred to Chapter 6.5 of Ref. [14].

[11] Note that term tensor is used loosely here to mean an array-like object with elements specified by an arbitrary number of indices, which is not the precise mathematical definition of a tensor.

B Additional Technical Details About CNNs

B.1 Zero-Padding

As was shown visually in Figures 12 and 13, the convolution process shrinks the input image along the spatial dimensions. In cases where this is undesirable, the input image is *zero-padded*—zeros are appended to its border, artificially increasing the input's size, which in turn increases the size of the output. In practice, zero padding is often used to ensure a convolution's input and output have equal spatial dimensions.

B.2 Stride

Often, one might wish to convolve the kernel and image at, say, every other position in the image. In this case, the kernel slides across the input image multiple pixels at a time; the number of pixels between successive kernel positions is called *stride*. Strides larger than one are equivalent to downsampling the input image, and are used to save on memory and computation cost for high-resolution images. For the interested reader, Ref. [11] provides intuitive visualizations and clear explanations of both zero-padding and stride.

B.3 Convolutional Arithmetic

We now present the (straightforward) formalism describing how an inputted image propagates through the layers of a convolutional network. Like in Appendix A for FCNs, we begin by introducing the notation needed to describe a CNN.

- Let C_{in} and C_{out} denote the number of channels in an input and output feature map, respectively.
- Let I_{w} and I_{h} denote the number of pixels in an input feature map along the width and height axes, respectively.
- Let K_{w} and K_{h} denote the number of pixels in a convolutional kernel.
- Let O_{w} and O_{h} denote the number of pixels in an output feature map.
- Let p_{w} and p_{h} (p for "padding") denote the number of zeros added to the beginning and end of the input feature map's width and height axes, respectively.
- Let s_{w} and s_{h} denote the kernel's stride along the width and height axes.

As is straightforwardly seen in Figure 12, the output shape O_j along the axis $j \in \{\text{w}, \text{h}\}$, for input and kernel dimensions I_j, K_j and for $s_j = 1$ and $p_j = 0$, is given by

$$O_j = (I_j - K_j) + 1.$$

More generally [11], the output shape along the axis j for arbitrary I_j, K_j, p_j and s_j is

$$O_j = \left\lfloor \frac{I_j + 2p_j - K_j}{s_j} \right\rfloor + 1, \tag{10}$$

where the vertical bars denote the floor function, used in case the stride s_j does not evenly divide the numerator.

We can now fully describe the propagation of an input feature map through a convolutional layer. Suppose the layer's input feature map has the shape $(C_{\text{in}}, I_{\text{w}}, I_{\text{h}})$, and a set of C_{out} kernels with spatial dimensions $(K_{\text{w}}, K_{\text{h}})$ is used to process the input image. In this case, the convolutional layer's output has the shape $(C_{\text{out}}, O_{\text{w}}, O_{\text{h}})$, where the output dimensions O_j are found from Equation 10. This output can then be used as the input to a subsequent convolutional layer with a different set of kernels, and the convolution process repeats in the next layer, as outlined in Section 4.3 on CNN architecture.

References

[1] M. Andrews, M. Paulini, S. Gleyzer, and B. Poczos,
End-to-End Physics Event Classification with CMS Open Data: Applying Image-Based Deep Learning to Detector Data for the Direct Classification of Collision Events at the LHC,
Computing and Software for Big Science **4** (2020), ISSN: 2510-2044,
URL: http://dx.doi.org/10.1007/s41781-020-00038-8.

[2] P. Baldi, P. Sadowski, and D. Whiteson,
Searching for exotic particles in high-energy physics with deep learning,
Nature Communications **5** (2014), ISSN: 2041-1723,
URL: http://dx.doi.org/10.1038/ncomms5308.

[3] CMS Collaboration, *Detecting Muons | CMS Experiment*, 2020,
URL: https://cms.cern/detector/detecting-muons (visited on 05/09/2021).

[4] CMS Collaboration, *Simulated dataset DiPhotonBorn_Pt-25To250_8TeV_ext-pythia6 in AODSIM format for 2012 collision data*, 2017, URL: http://opendata.cern.ch/record/7687.

[5] CMS Collaboration,
Simulated dataset GJet_Pt40_doubleEMEnriched_TuneZ2star_8TeV_ext-pythia6 in AODSIM format for 2012 collision data, 2017, URL: http://opendata.cern.ch/record/7778.

[6] CMS Collaboration, *Simulated dataset GluGluHToGG_M-125_8TeV-pythia6 in AODSIM format for 2012 collision data*, (2017), URL: http://opendata.cern.ch/record/7783.

[7] CMS Collaboration, *The CMS electromagnetic calorimeter project: Technical Design Report*, tech. rep., 1997, URL: https://cds.cern.ch/record/349375.

[8] CMS Collaboration,
The CMS experiment at the CERN LHC. The Compact Muon Solenoid experiment,
JINST **3** (2008) S08004. 361 p, URL: https://cds.cern.ch/record/1129810.

[9] CMS Collaboration, *The CMS hadron calorimeter project: Technical Design Report*, tech. rep., 1997, URL: https://cds.cern.ch/record/357153.

[10] CMS Collaboration, *The CMS tracker system project: Technical Design Report*, tech. rep., 1997, URL: https://cds.cern.ch/record/368412.

[11] Vincent Dumoulin and Francesco Visin, *A guide to convolution arithmetic for deep learning*, ArXiv e-prints (2016), eprint: 1603.07285, URL: https://arxiv.org/abs/1603.07285.

[12] Lyndon R Evans and Philip Bryant, *LHC Machine*, JINST **3** (2008) S08001. 164 p,
URL: https://cds.cern.ch/record/1129806.

[13] Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 2st, O'Reilly Media, Inc., 2019, ISBN: 1491962291.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016,
ISBN: 9780262035613, URL: http://www.deeplearningbook.org.

[15] Julie Haffner, *The CERN accelerator complex*, General Photo, 2013,
URL: https://cds.cern.ch/record/1621894.

[16] J. G. Layter, *The CMS muon project: Technical Design Report*, tech. rep., 1997,
URL: https://cds.cern.ch/record/343814.

[17] Tai Sakuma, *Cutaway diagrams of CMS detector*, 2019,
URL: https://cds.cern.ch/record/2665537.

[18] A.M. et al Sirunyan,
Particle-flow reconstruction and global event description with the CMS detector., 2017,
arXiv: 1706.04965, URL: https://cds.cern.ch/record/2270046.

[19] Maurizio Vretenar et al., *Linac4 design report*, tech. rep., 2020,
URL: https://cds.cern.ch/record/2736208.