# Chapter 5

# Machine Learning for Analysis and Instrumentation in High Energy Physics

Javier M. Duarte* and Dylan S. Rankin†

*University of California San Diego, La Jolla, CA 92093, USA
†University of Pennsylvania, Philadelphia, PA 19104, USA

## 5.1 Introduction

In high energy physics (HEP), the study and use of machine learning (ML)—the practice of solving problems by allowing machines to "discover" algorithms using data or experience without explicit programming—have been exploded in recent years. According to the INSPIRE HEP database, the number of articles in HEP and related fields that refer to ML and related topics has grown twenty times compared to ten years ago.[1] Notwithstanding this recent surge of interest, ML has deep ties to HEP, especially instrumentation, with early work dating back to the late 1980s and early 1990s [31, 33–35, 87–89]. In these early days, the most popular techniques, including cellular automata and multi-layer perceptrons, helped shape experimental particle physics. As deep neural networks have achieved human-level performance for various tasks, such as image classification [59, 79] in the early 2010s, they were adopted more regularly in particle physics [14, 56, 61, 106]. Unlike traditional approaches, deep learning techniques operate on lower-level information to extract higher-level patterns directly from the data.

 https://doi.org/10.1142/9789819801107_0005

[1] https://inspirehep.net/literature?q=%28deep%20learning%29%20OR%20%28neural%20network%29%20OR%20%28machine%20learning%29%20OR%20%28artificial%20intelligence%29.


Figure 5.1. Nomological net of topics in ML in particle physics inspired by the HEP ML Living Review [61].

ML in particle physics has become more than a tool and has emerged as a subfield worthy of intense academic study in its own right. This can be seen through the HEP ML Living Review [61], which as of January 2024[2] categorizes 1,252 articles, proceedings, reviews, book chapters, and other contributions in this subfield. Inspired by this classification, we can visualize the different topics of ML in particle physics as a *nomological net* in Fig. 5.1. Use cases range from standard classification and regression to simulation, uncertainty quantification, and real-time inference.

[2]https://github.com/iml-wg/HEPML-LivingReview/blob/2c7cd26/HEPML.bib.

This chapter is meant to introduce the reader to the basic concepts of ML that are widely used in HEP. After reviewing these concepts, we survey popular applications in HEP.

## 5.2 Machine Learning Basics

### 5.2.1 *Types of learning*

The basic premise of ML is to use a set of observations to uncover an underlying process corresponding to an unknown target function mapping the inputs to the correct outputs. Within this framework, there are several different types of learning paradigms, which differ in the information contained in the dataset and how that information is used. When observations are coupled with correct outputs, known as *labels*, based on reliable information from simulation or empirical observation (ground truth), and the learning process that uses them, this is known as *supervised learning*. This is the most prevalent and well-studied form of learning in HEP and beyond, but other types are increasingly being applied. For example, in *unsupervised learning*, the training data do not contain any desired output or label information at all. For the remainder of this chapter, we focus primarily on supervised learning, but we discuss some applications of unsupervised learning.

### 5.2.2 *Supervised learning*

Within supervised learning, different tasks require different types of outputs. Tasks that require producing continuous, real-valued predictions, for example for quantities like mass, temperature, or energy, are known as *regression*. On the contrary, the main goal of *classification* is to assign, among a set of fixed options, the category to which a data sample belongs. Typically, the output of the model is a set of values $p_i \in [0, 1]$, one for each class, that represent the probabilities that the data sample belongs to a particular class $i$.

Given a training dataset $S = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ consisting of data samples in an input domain $x_i \in \mathcal{X}$ and labels in a target domain $y_i \in \mathcal{Y}$, where $i$ indexes the sample in the dataset, the goal is to learn a function from the input to the output domain $f\colon \mathcal{X} \to \mathcal{Y}$, parameterized by a vector of *parameters* $\theta$, that best approximates the labels. We denote the output of the function for a given input $x$ as $f(x|\theta)$. The space of functions under consideration is known as the *model or hypothesis class*.

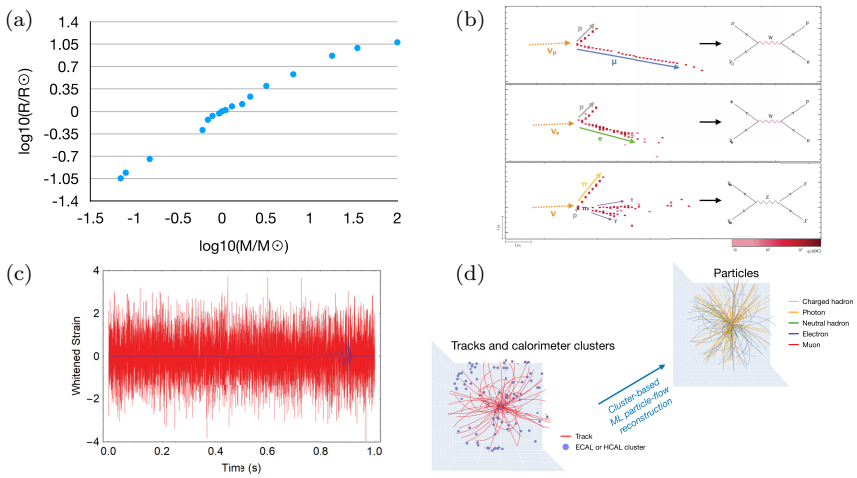


Figure 5.2. Examples of data representations and supervised learning tasks in physics, including (a) predicting the mass of a star given a measurement of its radius, (b) classifying image data from the NOvA experiment as one of the four types of neutrino interactions [6], (c) reducing noise in time series data to better identify gravitational wave signals [101], and (d) reconstructing particles based on detector measurements in a collider experiment [103, 104].

Examples of supervised learning are illustrated in Fig. 5.2:

(a) Predicting the mass of a star given a measurement of its radius. In this case, the input domain corresponds to the set of real numbers $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$.

(b) Classifying image data from the NOvA experiment as one of four types of neutrino interactions [6]. In this case, $\mathcal{X} = \mathbb{R}^{100\times80\times2}$ because there are two detector views ($x - z$ and $y - z$) with each image featuring 100 by 80 pixels of information. The target domain is a set of labels $\mathcal{Y} = \{\nu_\mu \text{ CC}, \nu_e \text{ CC}, \nu_\tau \text{ CC}, \nu \text{ NC}\}$, where each element is a different type of neutrino interaction.

(c) Reducing noise in time series data to better identify gravitational wave signals [101]. For this task, $\mathcal{X} = \mathbb{R}^{8192}$ and $\mathcal{Y} = \mathbb{R}^{8192}$, corresponding to 8 s of the data sampled at a rate of 1024 Hz, before and after noise reduction.

(d) Reconstructing particles based on detector measurements in an LHC experiment [103]. Here, $\mathcal{X} = \prod_{i=1}^{6,400} \mathbb{R}^7 \times \{\text{track}, \text{cluster}\} \times \{-1, 0, +1\}$, where there are seven continuous features and two discrete features

(whether the measurement is a calorimeter cluster or a track and the measured charge of the track) for up to 6,400 measurements per event. The target domain $\mathcal{Y} = \prod_{i=1}^{6,400} \mathbb{R}^4 \times \{\text{charged hadron}, \text{neutral hadron}, \gamma, e^{\pm}, \mu^{\pm}\} \times \{-1, 0, +1\}$ because there are four continuous features (four momentum of the particle) and two discrete features (particle type and charge) for up to 6,400 particles per event.

### 5.2.3 *Objective function*

The *objective function*, often called the *loss* or *cost function*, $L(y_i, f(x_i|\theta))$ measures the quality of predictions made by an ML algorithm. For example, a simple choice for regression problems is the squared loss $L(y, y') = (y' - y)^2$. The farther away the predicted value $y'$ is from the true value $y$, the larger the value of the loss function. The more accurate an ML algorithm is, the smaller the loss value should be, on average, for a given set of data. Therefore, our goal is to minimize the loss function.

The *learning objective* is to find the parameters that minimize the loss function averaged over the entire training dataset, which we denote $l(\theta)$. These optimal parameters, denoted $\theta^*$, can be expressed using the arg min operator, which returns the value where a given function attains its minimum:

$$\theta^* = \arg\min_{\theta} l(\theta) \equiv \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i|\theta)) \tag{5.1}$$

Roughly speaking, $\theta^*$ is the set of parameters that minimizes the difference of the output of the algorithm and ground truth label.

Depending on the type of optimization process, there are additional requirements for the loss function. For example, the *gradient descent* algorithm discussed in Section 5.2.6 requires calculating the gradient of the loss function with respect to the model parameters to determine how to modify the parameters to reduce the loss function. Thus, the loss function must be *differentiable* in the model parameters.

Training an ML algorithm is closely related to statistical inference via the method of maximum likelihood [49]. In the maximum likelihood method, observed data are modeled by a probability distribution function with some free parameters. To estimate those parameters, we find their values such that the observed data are the most probable under this statistical model.

There is a correspondence between commonly used loss functions and likelihood functions. For example, minimizing the squared loss corresponds to maximizing a Gaussian likelihood. A Gaussian likelihood with observed value $y'$ and expected mean $y$ and standard deviation $\sigma_y$ is given by

$$G(y'|y,\sigma_y) = \frac{1}{\sigma_y\sqrt{2\pi}} \exp\left(\frac{-(y'-y)^2}{2\sigma_y^2}\right) \tag{5.2}$$

If we take the negative logarithm of this likelihood,

$$-\ln G(y'|y,\sigma_y) = (y'-y)^2/(2\sigma_y^2) + \ln(\sigma_y\sqrt{2\pi}) \tag{5.3}$$

$$= c(y'-y)^2 + b \tag{5.4}$$

we see that up to a multiplicative constant $c$ and an additive constant $b$, this is equivalent to the squared loss.

Another common loss function appropriate for binary classification tasks is the binary cross-entropy (BCE), which can be derived from the Bernoulli likelihood. Given two true classes, $y = 0$ or $y = 1$, and a model output $y'$ defined between 0 and 1, which represents the probability that the data sample belongs to the $y = 1$ class, the Bernoulli distribution defines the likelihood

$$B(y'|y) = (y')^{\delta[y=1]}(1-y')^{\delta[y=0]} \tag{5.5}$$

where the $\delta$ operator evaluates to 1 or 0 if the argument is true or false, respectively. Note that only one of these two terms appears, depending on the true value of $y$. Taking the negative logarithm of the likelihood yields the BCE loss function:

$$L_{\text{BCE}}(y,y') = -\ln B(y'|y) = -\delta[y=1]\ln y' - \delta[y=0]\ln(1-y') \tag{5.6}$$

This can also be generalized to the categorical cross-entropy (CCE) for classification tasks with more than two target classes.

Figure 5.3 compares the squared loss and BCE for a given example whose true label is $y = 0$. Although both losses increase the farther $y'$ is from $y$, BCE is more appropriate for classification problems because it takes into account that $y' = 1$ is an extremely incorrect prediction and the loss grows without bound as $y' \to 1$.

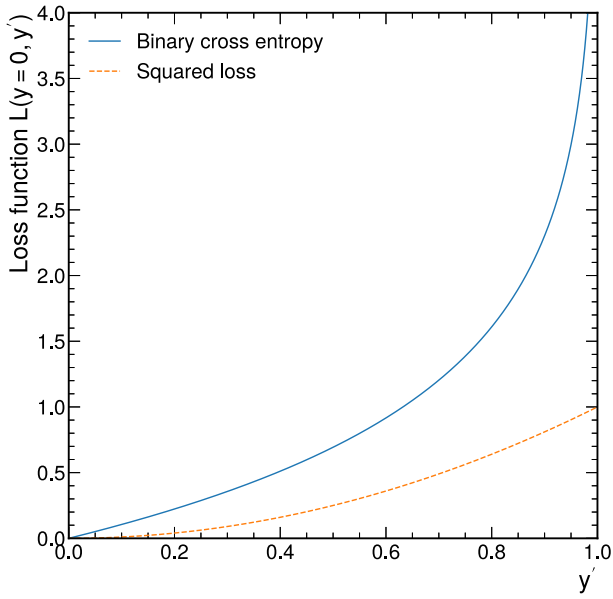


Figure 5.3. Comparison of the squared and binary cross-entropy loss functions for a true value of $y = 0$. The BCE loss grows without bound as the prediction $y'$ approaches 1.

### 5.2.4 *Linear models*

Despite their simplicity, linear models are the workhorse of machine learning. Given a set of $D$ features, each data point is a vector in $D$-dimensional space $x \in \mathbb{R}^D$, and a linear model can be expressed as

$$f(x|\theta, b) = \theta^\intercal x + b \tag{5.7}$$

where the *weight* $\theta \in \mathbb{R}^D$ and *bias* $b \in \mathbb{R}$ are unconstrained parameters of the model. These parameters are chosen to minimize the loss function on the training data. For notational convenience, we can absorb the bias into the weight vector by extending the input vector with a constant feature $x^{(0)} = 1$ and setting the corresponding entry of the weight vector equal to the bias $\theta^{(0)} = b$. This allows us to express linear models directly as

$$f(x|\theta) = \theta^\intercal x \tag{5.8}$$

To expand on the foundational ML concepts, we introduce an explicit example of regressing the logarithm of the radius of stars in the so-called main sequence as a function of the logarithm of the star mass. This means

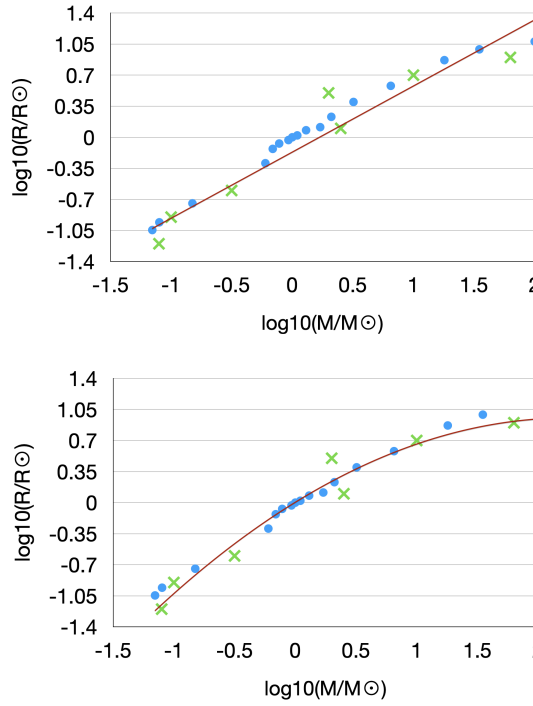


Figure 5.4. Training data points (crosses), testing data points (dots), and linear models (lines) fit to the data. A linear model using the original features $x$ (upper) and a linear model after using a polynomial embedding $\phi(x) = (1, x, x^2)$ (lower) are shown.

we will train a model to predict $\log_{10}(R/R_\odot)$ given $\log_{10}(M/M_\odot)$. Sample data, split into training data (crosses) and testing data (dots), and a trained linear model (line) are shown in Fig. 5.4 (upper).

Linear models can perform more challenging tasks by replacing our input $x$ with a transformation or *embedding* of $x$ called $\phi(x)$. To illustrate this, consider a classification task in which we want to separate the two classes of data points in the $(x_1, x_2)$ plane, represented by $+$ and $\circ$ symbols, respectively, as shown in Fig. 5.5 (left). The two classes could be separated by a circular boundary. Unfortunately, linear models can only create boundaries that are straight lines. Thus, no linear model can perfectly separate these two classes of data in the original input space of $(x_1, x_2)$. However, we can apply a simple transformation squaring both components of the input $\phi(x_1, x_2) = (x_1^2, x_2^2)$, as shown in Fig. 5.5 (lower). Now, the two

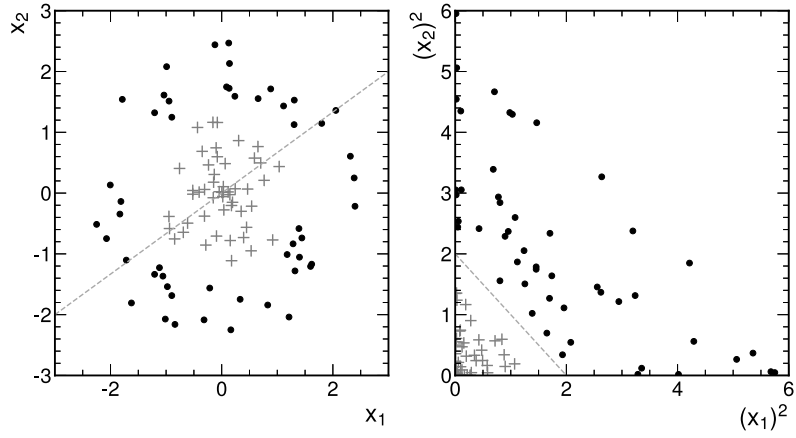


Figure 5.5. Example of embedding for a classification task. Two classes of data points in the $(x_1, x_2)$ plane, represented by + and ∘ symbols, respectively, cannot be separated by a straight line (left). After transforming the data $\phi(x_1, x_2) = (x_1^2, x_2^2)$, the two classes can be separated by a straight line (right).

classes are separable by the straight line shown, which we can implement with a linear model.

More quantitatively, we can return to our regression task. If we use a polynomial embedding $\phi(x) = (1, x, x^2)$, then our model becomes

$$f(\phi(x)|\theta) = \theta^{\mathsf{T}}\phi(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \tag{5.9}$$

This model achieves a smaller training error than a linear model with the original feature $x$, as shown in Fig. 5.4 (lower). We say that this model is *more expressive* because it can represent a wider variety of functions. Although this is equivalent to polynomial regression in the original feature $x$, it is still a linear model in the new embedded features $\phi(x)$. For certain models, it is even possible to use the discriminating power of the embedded features without explicitly calculating them through the so-called "kernel trick." Further discussion of kernel methods can be found in Hofmann *et al.* [67], Scholkopf and Smola [114].

Although we have not yet defined neural networks, we can already try to build some intuition for how they work based on the concepts discussed already. As shown in Fig. 5.6, neural networks have linear models as their basic building block. A neural network can be thought of as a linear model after inputs are mapped to features through a nonlinear transformation. The initial layers of a neural network act as "automatic featurizers," where

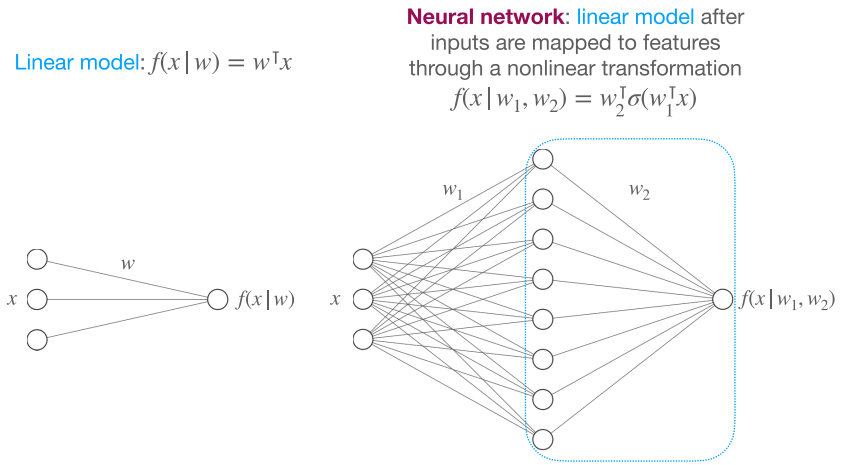


Figure 5.6. More complex model classes like neural networks have linear models as their basic building block. A neural network can be thought of as a linear model after inputs are mapped to features through a nonlinear transformation. Neural networks are "automatic featurizers."

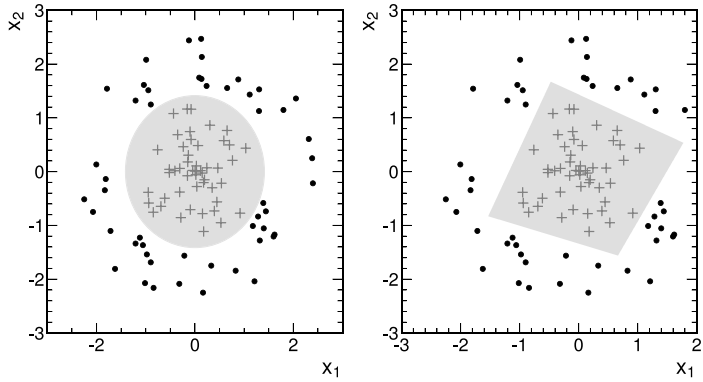


Figure 5.7. Decision boundary of a linear model after an embedding $\phi(x_1, x_2) = (x_1^2, x_2^2)$, corresponding to $x_1^2 + x_2^2 = 2$ (left). Decision boundary for a simple two-layer neural network with three hidden features (right).

instead of us guessing a well-suited embedding of our input features, the model learns one directly.

Revisiting the classification task of Fig. 5.5, a simple two-layer neural network can map the two input features to three "hidden features" where the two classes are separable. Figure 5.7 displays the decision boundaries

for the linear model after the embedding described previously and a simple neural network. Since the embedding is hand-tuned for this dataset, its decision boundary can be thought of as ideal. The neural network's decision boundary is an imperfect approximation with jagged corners, but it has the advantage that no feature engineering was necessary—the features were learned by the neural network automatically. To gain intuition for neural networks before we describe them fully in Section 5.3.2, readers are encouraged to explore a visualization tool called TensorFlow Playground at https://playground.tensorflow.org.

### 5.2.5 *Generalization and bias-variance decomposition*

One of the central goals of ML is to train models that *generalize*, meaning that they perform well on test data outside the training set. But what exactly does that mean? Generally, it means the *expected test error* is small. As we see, two main sources of error prevent ML algorithms from generalizing beyond their training set. One is *bias* arising from erroneous assumptions in the ML algorithm and the other is *variance* arising from sensitivity to statistical fluctuations in the training set. A graphical visualization of bias and variance is shown in Fig. 5.8.

These ideas are connected to *underfitting*, when a model is unable to capture the relationship between the inputs and labels accurately, resulting in a large error rate in both training and test data, and *overfitting*, when a model fits exactly (or nearly so) in training data but does not perform accurately on test data. Explicit examples of both underfitting and overfitting are shown in Fig. 5.9. In this case, either a zeroth-order (upper) or fifth-order polynomial (lower) is used to fit the training data. The zeroth-order polynomial underfits the training data, resulting in a large

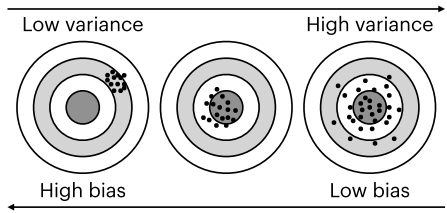


Figure 5.8. Graphical visualization of bias and variance using a bulls-eye diagram. Each hit represents a different, individual training of an ML model. The proximity to the center of the bulls-eye target indicates how low the test error is. Three different cases representing different combinations of high and low bias and variance are shown.

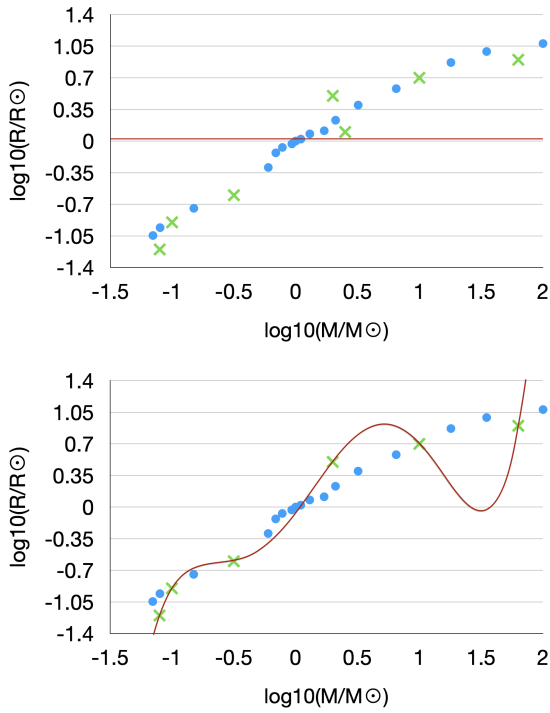


Figure 5.9. Examples of underfitting with a zeroth-order polynomial (upper) and overfitting with a fifth-order polynomial (lower). Training data points (crosses), testing data points (dots), and models (solid lines) fit to the data.

test error due to its high bias. Correspondingly, the fifth-order polynomial overfits the training data, also resulting in a large test error due to its high variance.

The *bias-variance decomposition* is a way of analyzing an ML algorithm's expected test error as a sum of bias and variance terms. To formalize this concept, we must introduce some statistical concepts and notation. For a random variable $x$ sampled from a probability density function (PDF) $P(x)$, which we denote $x \sim P(x)$, we can define its expected value as

$$\mathbb{E}_{x\sim P(x)}[x] = \int_{-\infty}^{\infty} x' P(x') dx'. \tag{5.10}$$

The expectation operator $\mathbb{E}$ is a generalization of the weighted average, where a subscript usually denotes the random variable(s) being sampled. Informally, the expected value is the arithmetic mean of a large number

of independently selected outcomes of a random variable. For a continuous random variable, we effectively weight the integral by the PDF. For an integrable function $f(x)$ of the random variable, we can obtain its expected value in an analogous way:

$$\mathbb{E}_{x\sim P(x)}[f(x)] = \int_{-\infty}^{\infty} f(x')P(x')dx' \tag{5.11}$$

Returning to the question of the generalizability of our models, we examine the test error. Assuming each training data point $(x_i, y_i)$ is sampled independently from $P(x, y)$ the "true" unknown probability distribution, then a trained model $f(x|\theta)$ has a true test error

$$L_P(f) = \mathbb{E}_{(x,y)\sim P(x,y)}\left[L(y, f(x|\theta))\right] \tag{5.12}$$

In general, we cannot compute this quantity, but we can estimate it using a test set of independent samples from $P(x, y)$. The training error is generally smaller than the test error. Overfitting occurs when the test error is much larger than the training error, while underfitting corresponds to the case when the training and test error are similar, but both are high.

The optimal set of model parameters $\theta_S^*$ is a function of the training dataset $S$. We can rewrite Eq. (5.1) to make this dependence explicit,

$$\theta_S^* = \arg\min_{\theta} \frac{1}{|S|} \sum_{(x,y)\in S} L(y, f(x|\theta)) \tag{5.13}$$

that is, if we change the training dataset $S$, the optimal set of parameters may change as well. The optimal parameters $\theta_S^*$ are themselves random variables because the training dataset $S$ is randomly sampled.

We can write the expected test error over all possible training datasets as

$$\mathbb{E}_S\left[L_P(f(x|\theta_S)\right] = \mathbb{E}_S\left[\mathbb{E}_{(x,y)\sim P(x,y)}\left[L(y, f(x|\theta_S))\right]\right] \tag{5.14}$$

If $L$ is the squared loss, we leave it as an exercise to the reader to show that we can decompose this expected test error into two terms:

$$\mathbb{E}_S\left[L_P(f(x|\theta_S)\right] = \mathbb{E}_{(x,y)\sim P(x,y)}\left[\underbrace{\mathbb{E}_S\left[(f(x|\theta_S) - F(x))^2\right]}_{\text{variance}} + \underbrace{(F(x) - y)^2}_{\text{bias}}\right] \tag{5.15}$$

where $F(x) \equiv \mathbb{E}_S\left[f(x|\theta_S)\right]$ can be thought of as the "average" prediction of our model over different possible training datasets.

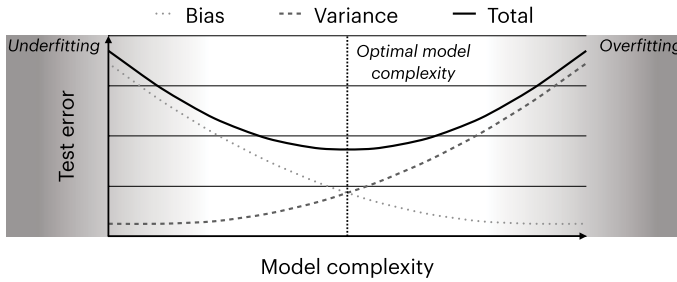


Figure 5.10. Bias-variance decomposition of test error as a function of model complexity.

How can we interpret Eq. (5.15)? The first term inside the expectation operator quantifies the *variance*: the difference in predictions when training on different datasets. The second term quantifies the *bias*: the difference of the average prediction from the ground truth. Thus, there is naturally a tradeoff: models with high variance tend to have low bias and vice versa.

We can relate overfitting and underfitting to the concepts of bias and variance. Overfitting implies high variance: the model class is too complex and retraining yields vastly different models. Variance tends to increase with model complexity and decrease with more training data. Underfitting implies high bias: the model class is too simple and has a large error rate. This relationship is shown schematically in Fig. 5.10.

### 5.2.6 *Optimization*

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. It is the basis for many of the optimization algorithms commonly used in modern ML. "First order" means it only requires first derivatives of the function. The idea is to start with some (possibly random) initial values for all the parameters and then compute the gradient of the function with respect to all the parameters. The gradient represents the direction of the steepest ascent of the function in parameter space. Since we want to minimize the function, we take a small step in the opposite direction of the gradient by updating the parameter values. Then, we repeat this process until we reach a minimum.

More precisely, the gradient descent algorithm proceeds as follows. Each *iteration* of the algorithm is indexed by an integer $t$, starting with $t = 0$, and the current values of the parameters are $\theta_t$. We set the parameters to some initial values, for example $\theta_0 = 0$ or randomly sampled from a Gaussian

distribution $\theta_0 \sim \mathcal{N}(\mu = 0, \sigma = 1)$ or some other distribution specific to a particular type of learning algorithm. At iteration $t$, the parameters are updated using the negative of the loss function gradient:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta l(\theta_t) \tag{5.16}$$

$$= \theta_t - \frac{\eta}{N} \nabla_\theta \sum_{i=1}^{N} L(y_i, f(x_i|\theta_t)) \tag{5.17}$$

where $\eta$ is a hyperparameter known as the step size or *learning rate*. The learning rate controls how large a step the algorithm takes during each update.

Unfortunately, we cannot determine *a priori* the optimal learning rate for a given model on a given dataset. Instead, a good (or good enough) learning rate must be discovered through trial and error. Typical values to consider are in the range of $\eta \in [10^{-6}, 1]$, while a good starting point is generally $10^{-3}$ or $10^{-2}$. If you set the learning rate too high, your training may not converge because the weight updates "overshoot" the minimum of the loss function. If you set the learning rate too high, your model may also not converge (or converge too slowly) because the weight updates are tiny. Hyperparameter optimization procedures, like grid search, Bayesian optimization, or the asynchronous successive halving algorithm [84], can help find a good learning rates.

Note that in Eq. (5.17), the entire "batch" of training data is used to determine the gradient. In principle, this can give a more accurate estimate of the test loss that is less susceptible to statistical fluctuations, at the cost of more computation, that is, iterating over the full training dataset for each update. We repeat these updates until we reach some predefined convergence criteria.

A popular variant of this algorithm is *stochastic gradient descent (SGD)*. In this case, the true gradient over the entire dataset is approximated by that for a single data point. In other words, the update rule is modified to consider only one, usually shuffled, data point $(x_i, y_i)$ at a time:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(y_i, f(x_i|\theta_t)) \tag{5.18}$$

Although this is much more computationally efficient, it can be subject to large statistical fluctuations.

At this point, it may be helpful to work through an end-to-end example of SGD for a regression problem, as shown in Fig. 5.11. Consider a training dataset consisting of two labeled data points $(x_1 = (1, 1), y_1 = 1)$ and

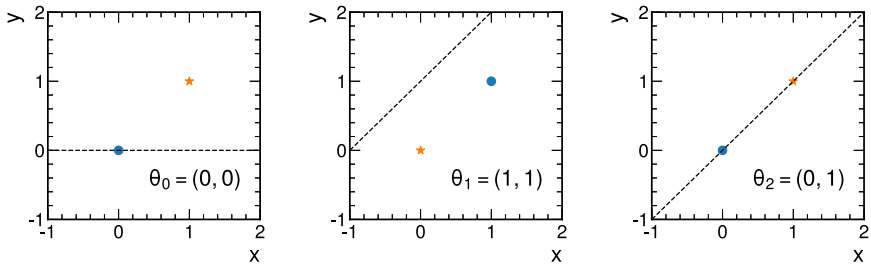


Figure 5.11. Example of stochastic gradient descent with two data points. Each frame from left to right represents an SGD iteration. The dotted line represents the current model with the current parameters listed on the canvas. The starred data point represents the one being used to compute the next parameter update. SGD converges after the second iteration.

$(x_2 = (1,0), y_2 = 0)$, where we have augmented the input with the "dummy" feature of 1 to simplify notation as described earlier. We use the squared loss function, a learning rate of $\eta = 0.5$, and an initial set of parameters $\theta_0 = (0,0)$, which includes the bias as the first component.

First, we can calculate the gradient of the loss with respect to the parameters:

$$\nabla_\theta L(y, f(x|\theta)) = \nabla_\theta (y - \theta^\intercal x)^2 = -2(y - \theta^\intercal x)x \tag{5.19}$$

Now we can write the SGD update rule of Eq. (5.18) as

$$\theta_{t+1} = \theta_t + 2\eta(y - \theta_t^\intercal x)x \tag{5.20}$$

$$= \theta_t + (y - \theta_t^\intercal x)x \tag{5.21}$$

where in the second line we use the fact that $\eta = 0.5$. Performing the first update with the data point $(x_1, y_1)$ yields

$$\theta_1 = \theta_0 + (y_1 - \theta_0^\intercal x_1) = \theta_0 + x_1 \tag{5.22}$$

$$= (0,0) + (1,1) = (1,1) \tag{5.23}$$

Similarly, the second update with the data point $(x_2, y_2)$ gives

$$\theta_2 = \theta_1 + (y_2 - \theta_1^\intercal x_2) = \theta_1 - x_2 \tag{5.24}$$

$$= (1,1) - (1,0) = (0,1) \tag{5.25}$$

which is exactly the optimal set of parameters. In this example, SGD converges after two iterations and will not give any further updates to the parameters because the loss is now zero for all data points, i.e., the data

are fit perfectly. We note that the example here was carefully chosen, and, in general, many more updates are required.

A compromise between batch and stochastic gradient descent is *mini-batch stochastic gradient descent*, where the gradient is approximated by the average over a mini-batch of $N_b$ samples:

$$\theta_{t+1} = \theta_t - \frac{\eta}{N_b} \nabla_\theta \sum_{i=1}^{N_b} L(y_i, f(x_i|\theta_t)) \tag{5.26}$$

This is more computationally efficient and may result in smoother convergence, as the gradient computed at each step is averaged over more training samples. The hyperparameter $N_b$ is known as the *mini-batch size*, which is typically taken to be a power of 2. It has been observed that choosing a large mini-batch size to train deep neural networks appears to deteriorate generalization [82]. One explanation for this phenomenon is that large mini-batch SGD produces "sharp" minima that generalize worse [64,74]. Specialized training procedures to achieve good performance with large mini-batch sizes have also been proposed [55,66,126].

Many alternatives to SGD have been developed to improve training dynamics and avoid common pitfalls, such as slow progress along shallow parameter dimensions, "jitter" or oscillations along steep parameter dimensions, sensitivity to parameter initialization, excessively noisy gradient estimates, and getting stuck in local or sharp minima. SGD with momentum, named by analogy with physical momentum, remembers previous updates in an attempt to accelerate training, reduce the impact of statistical fluctuations, and prevent getting stuck in local minima [100,110,118].

Adaptive momentum estimation (Adam) [76] is an extremely popular SGD variant that combines many improvements from its predecessors [44, 62,127] to make it more robust. In particular, it uses an adaptive learning rate specialized for each parameter. Figure 5.12 illustrates a comparison of SGD-based methods. Momentum can be seen as a ball running down a slope while Adam behaves like a heavy ball with friction that prefers flat minima in the error surface.

### 5.2.7 *Regularization*

Regularization refers to the practice of applying constraints, either implicitly or explicitly, to a model in order to guide optimization toward a simpler solution to prevent overfitting and improve generalization. As the complexity, capacity, and sheer number of parameters of ML models have

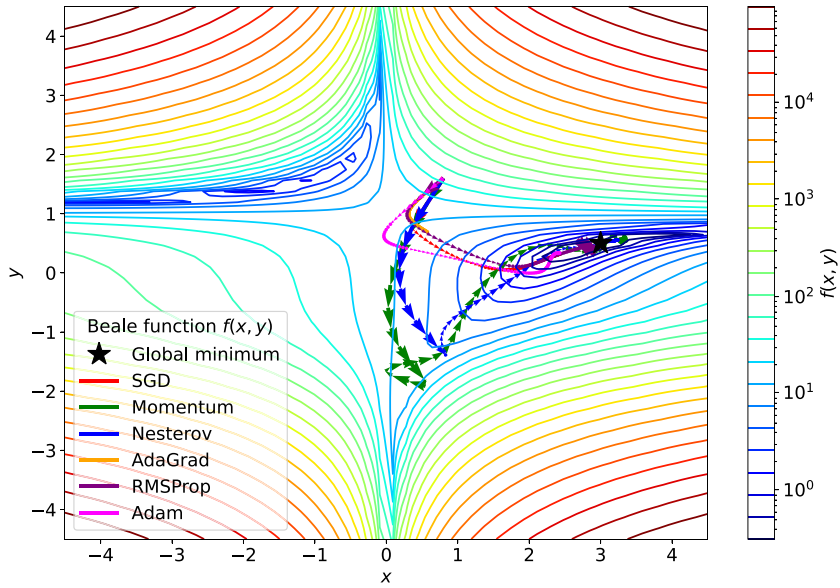


Figure 5.12. Comparison of different SGD methods optimizing the Beale function $f(x, y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2$ with global minimum $f(3, 0.5) = 0$.

grown in recent years, the likelihood of overfitting becomes greater, making regularization a critical component of modern ML. Explicit regularization refers to when an explicit term is added to the loss function, while implicit regularization includes other forms of regularization, for example, early stopping, using a robust loss function, and discarding outliers. Implicit regularization is ubiquitous in modern ML approaches, including stochastic gradient descent for training deep neural networks, and ensemble methods (such as random forests and gradient boosted trees).

The most common type of explicit regularization is $L_n$ regularization, in which a term is added to the loss that penalizes large weights and biases:

$$L_n = -\lambda_1 \sum_{i=1}^{N_\theta} |\theta_i|^n \tag{5.27}$$

where $\theta_i$ is parameter of the model. Usually, $n = 1$ (called $L_1$ regularization or lasso regression) or $n = 2$ (called $L_2$ regularization or ridge regression) is chosen. $L_1$ regularization naturally induces sparsity, whereas $L_2$ regularization tends to keep all parameters with lower magnitudes. The reason for this is illustrated in Fig. 5.13. In these two parameters, the constraint region for $L_1$ regularization is diamond-shaped, while for $L_2$, it is elliptical. Since

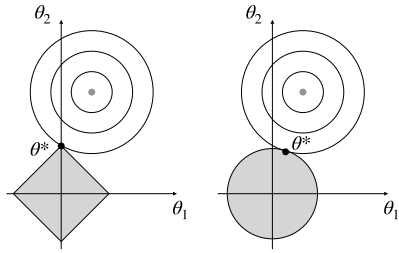


Figure 5.13. Depiction of $L_1$ (left) and $L_2$ (right) regularization constraint regions. The contours of an unregularized loss function are shown. The intersection with the constraint region from $L_1$ regularization gives an optimum value $\theta^*$ that is sparse, i.e., $\theta_1 = 0$. On the other hand, $L_2$ regularization yields an optimum value $\theta^*$ where both $\theta_1$ and $\theta_2$ are small but non-zero.

$L_1$ regularization sets certain weights to zero, it is often used as part of feature selection and model compression techniques. On the other hand, $L_2$ regularization reduces the contribution of high outlier nodes and distributes the weight given to correlated features, potentially leading to a more robust model.

A popular implicit regularization method is known as dropout [116], in which certain units are randomly dropped (along with their connections) from a neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, a single "unthinned" network is used that effectively averages the predictions of all these thinned networks. Dropout introduces a new hyperparameter $p$ (typically between 0.1 and 0.5) that specifies the probability of dropping units in a given layer.

To illustrate the effectiveness of regularization, we use a highly over-parameterized neural network (three hidden layers of 100 nodes each) to classify data generated according to spiral patterns, both with and without dropout ($p = 0.15$). The results are shown in Fig. 5.14. The unregularized network (left) overfits the data as the decision boundary encircles single data points. The regularized network (right) learns a decision boundary that is much more faithful to the underlying spiral pattern.

### 5.2.8 *Compression*

In recent years, ML models have grown dramatically in their computational complexity, from thousands of parameters and operations to millions or even billions. However, many real-world and HEP applications require real-time on-device processing capabilities. The main challenge is that the devices

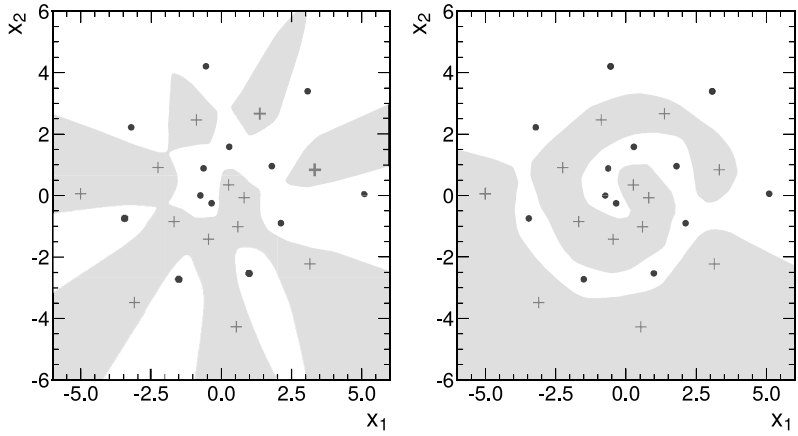


Figure 5.14. Decision boundary for a highly overparameterized network fitting spiral data with (right) and without (left) dropout ($p = 0.15$). The unregularized network (left) overfits the data as the decision boundary encircles single data points. The regularized network (right) learns a decision boundary that is much more faithful to the underlying spiral pattern.

used in these scenarios are resource-constrained, with limited memory, processing capabilities, and usually a strict latency budget. Reducing the size of ML models with *compression* can enable their use.

Compression techniques aim to improve the computational efficiency of models while keeping the performance as close as possible to the original. The two most ubiquitous methods are *quantization* [7, 30, 36, 41–43, 57, 70, 83, 92, 93, 95, 98, 107, 123, 128, 129, 131, 132], which modifies the number of bits used to calculate and store results in the model, and *pruning* [8, 46, 57, 81, 108, 130], which removes connections in a neural network.

In CPU- and GPU-based ML inference, it is common to use 32-bit floating-point precision. This allows the network to capture a very large range of values; the largest magnitude number that can be stored in 32-bit floating point format is $3.402823466 \times 10^{38}$ and the smallest is $1.175494351 \times 10^{-38}$. However, for many applications, the full floating-point precision range may not be required. Reduced-precision formats, such as integer or fixed-point precision, are commonly used instead, as shown in Fig. 5.15.

One disadvantage of reduced-precision formats with respect to floating point is a reduced dynamic range. Thus, care must be taken to ensure that weights or outputs of the ML model do not underflow or overflow in the

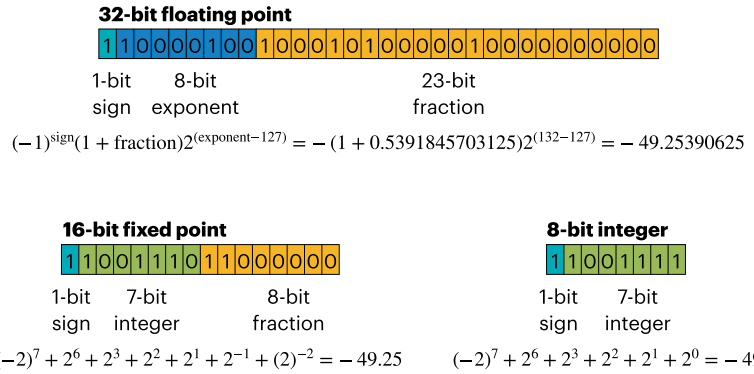


Figure 5.15. Comparison between 32-bit floating-point (upper), 16-bit fixed-point (lower left), and 8-bit integer (lower right) representations.

reduced-precision format. However, reduced-precision representations are much more amenable to computations on specialized hardware, such as field-programmable gate arrays (FPGAs).

We can distinguish *post-training quantization* (PTQ), in which model parameters are quantized after a traditional training is performed with 32-bit floating-point precision, and *quantization-aware training* (QAT), in which training is performed with a modified procedure designed to emulate reduced precision formats.

Pruning is the removal of unimportant weights, quantified in some way, from a neural network. The two main categories are *unstructured pruning*, where weights are removed without considering their location within a network, and *structured pruning*, where weights connected to a particular node, channel, or layer are removed. These are depicted in Fig. 5.16. Pruning reduces the number of computations that must be performed to produce an inference result, thus reducing the hardware resources or algorithm latency. There are many different ways to decide which connections can be removed in a network, and the development of pruning algorithms and understanding their behavior are active areas of research.

One relatively simple method of pruning weights is called iterative, magnitude-based pruning, illustrated in Fig. 5.17. In this process, an $L_1$ regularization term is added to the loss that penalizes large weights. Training with this loss term typically produces two populations in the weights for a given layer. The weights that are deemed unnecessary by

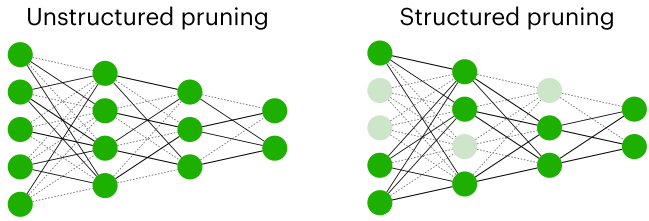


Figure 5.16. Pruning removes "unimportant" parameters and operations from a neural network. Removed connections are illustrated as gray dotted lines, while the remaining connections are solid black lines. Unstructured pruning (left) removes weights without considering their location within a network. Structured pruning (right) removes weights connected to a particular node, channel, or layer.

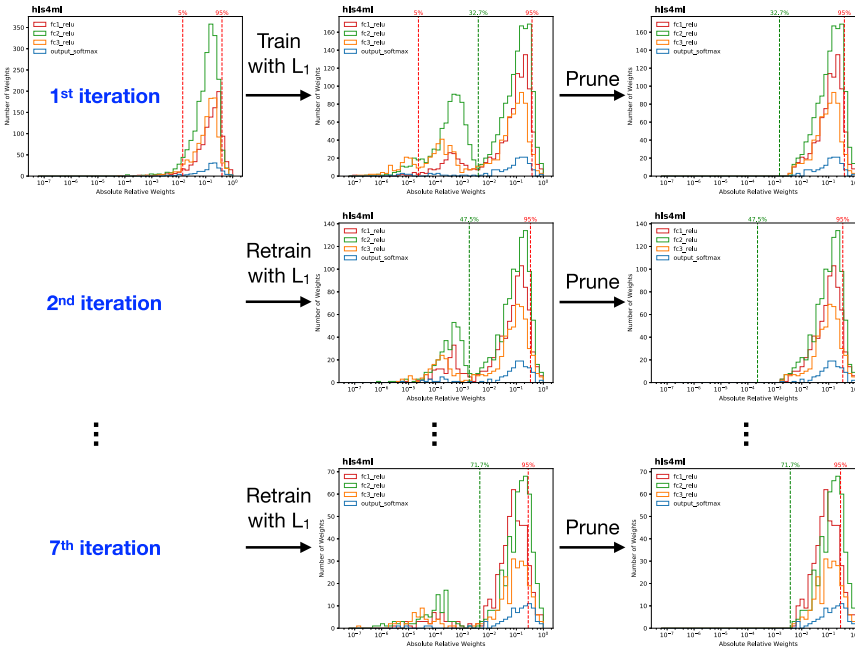


Figure 5.17. Illustration of the iterative magnitude-based parameter pruning and retraining with $L_1$ regularization procedure [43]. The distribution of the absolute value of the weights relative to the maximum absolute value of the weights is shown after each step of the pruning and retraining procedure. In the top left, the distribution before compression is shown, while in the bottom right, the distribution after compression is displayed.

the training will have very small values, while the weights that are deemed necessary will have larger values. Then, those weights with small values can be fixed to 0 (thereby removing that connection from the network), and training can be repeated. In many cases, successive training will identify additional weights that can be made small and thus removed. Repetition of this procedure can remove more weights until the desired reduction in connections, or *sparsity*, is achieved. This process usually results in networks that have slightly reduced performance compared to the full network, although the performance loss can be negligible depending on the target sparsity.

Both quantization and pruning can be applied together or individually depending on the problem at hand and implementation requirements, and the exact tradeoff between performance and sparsity or quantization is model-specific and depends on the model size, complexity, and task.

## 5.3 Models

In this section, we explore some of the most frequently used models in HEP.

### 5.3.1 *Decision trees*

Decision trees are among the simplest and most robust nonlinear models first invented in the context of data mining and pattern recognition as classification and regression trees (CART) [16]. Roughly speaking, they ask a series of yes-or-no questions based on individual features in order to categorize data. An example of a simple decision tree is shown in Fig. 5.18 to differentiate electron neutrino signal interactions ($\nu_e n \rightarrow pe^-$) from muon neutrino background interactions ($\nu_\mu n \rightarrow p\mu^-$) in the MiniBooNE detector [109]. In this case, the features used are relevant for this classification task, including the number of photomultiplier tube (PMT) hits, the total deposited energy, and the radius of the Cherenkov radiation ring. Distinguishing these two classes is essential to measure the quantum mechanical phenomenon of *neutrino oscillation*, in which a neutrino of one flavor (electron, muon, or tau) can later be measured to have a different flavor [53].

Formally, decision trees consist of a set of *internal*, or *branch*, *nodes*, that lead to two further nodes, and *terminal*, or *leaf*, *nodes* with no further branching. Every branch node $i$ has a binary query function $q_i(x)$ that maps the input $x$ to 0 or 1 and determines the subsequent node. The basic form

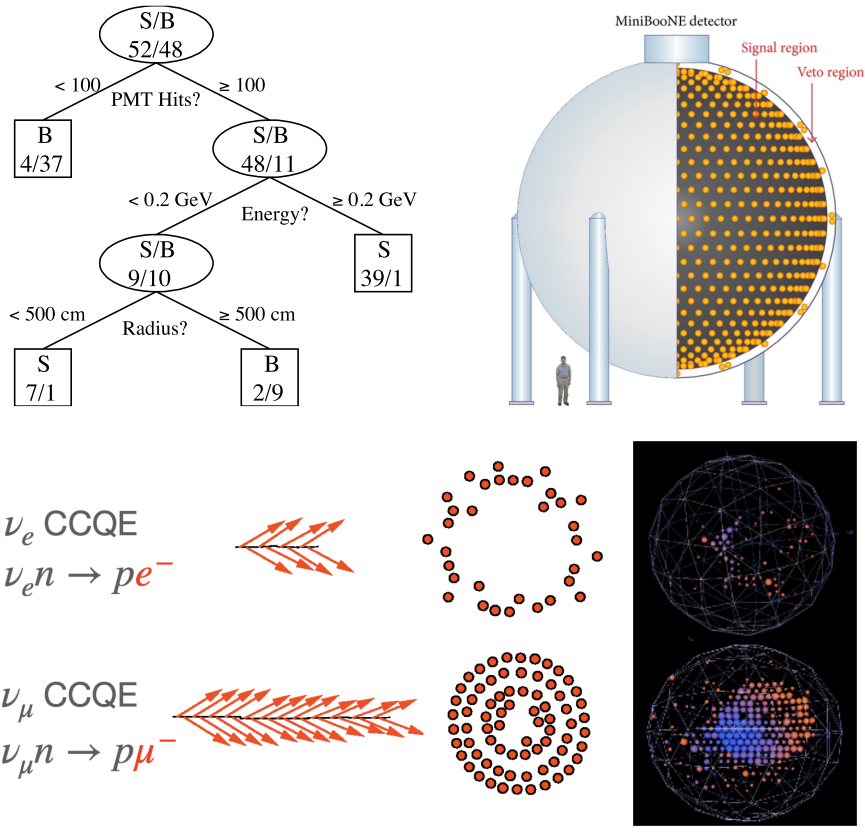


Figure 5.18. A schematic decision tree for event classification in the MiniBooNE experiment [109]. The goal is to differentiate signal $\nu_e n \to pe^-$ charged current quasi-elastic ($\nu_e$ CCQE) interactions from background $\nu_\mu n \to p\mu^-$ ($\nu_\mu$ CCQE) interactions based on the Cherenkov radiation patterns measured by the photomultiplier tubes.

of the query function is a cut in an individual component $x^{(d_i)}$ of $x$:

$$q_i(x) = \delta[x^{(d_i)} > c_i] \tag{5.28}$$

Every leaf node makes a constant prediction. For a given sample $x$, prediction begins at the root node, calling the query function for each visited node. If the returned value is 1, the left child node is chosen, while the right child node is chosen otherwise. This process is repeated until a leaf node is reached.

Decision trees express piecewise-constant functions. A given tree creates $J$ *axis-aligned* partitions of the input space $\mathcal{X} = \mathcal{X}^1 \bigcup \cdots \bigcup \mathcal{X}^J$, through a sequence of binary splits, where the length of the sequence is the *depth* of the tree. The number of leaf nodes is $J$. Each partition has a constant prediction $b_j$. The model can be written as

$$f(x|\theta) = \sum_j b_j \delta[x \in \mathcal{X}^j] \tag{5.29}$$

where $j \in \{1, \ldots, J\}$ indexes each leaf node.

Decision trees can often outperform linear models because they can learn nonlinear decision boundaries, as shown in Fig. 5.19 (upper). However, because most tree-based models consider splits aligned with individual feature components, there are some failure modes. In particular, it can be difficult to learn decision boundaries diagonally across two components, as shown in Fig. 5.19 (lower). Nonetheless, tree-based models are often preferred over other models because they work well with tabular data that

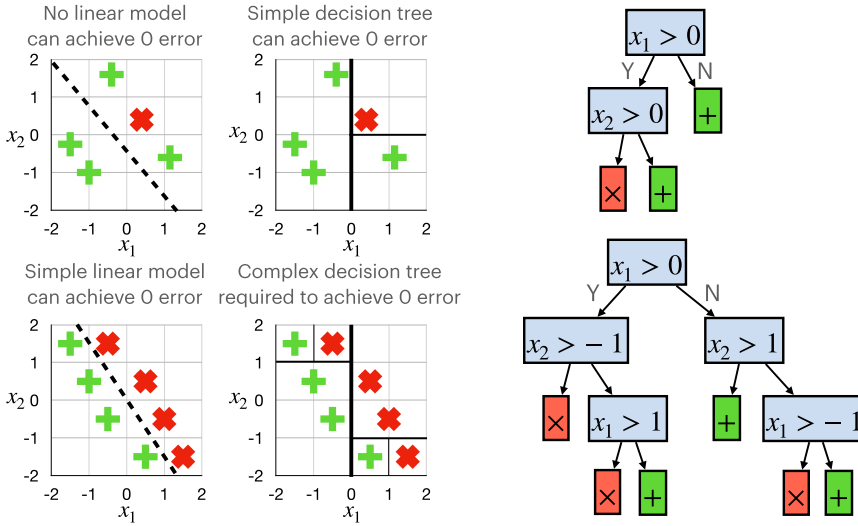


Figure 5.19. Two different cases demonstrating the strengths and weaknesses of linear models and decision trees. A decision tree can learn a nonlinear decision boundary unlike a linear model (upper). A simple linear model can learn a decision boundary diagonally across two feature components, while it requires a more complex decision tree to approximate the same decision boundary (lower).

may comprise a mix of continuous and discrete features, and there is less need for preprocessing.

So far we have described how decision trees look, but how are they constructed in the first place? A common (top-down) approach to building a decision tree starts with a root node and grows the tree with splits based on individual components of $x$. To decide when a given split is advantageous, we need to use a metric, called an *impurity measure*. Generally, they quantify to what degree a split refines the terminal nodes to be more pure than the parent node. The most widely used measure is the Gini impurity [16] defined as

$$I_{\text{Gini}} = \left(1 - p^2 - (1-p)^2\right) \tag{5.30}$$

where $p$ is the fraction of positive examples ($y = 1$) in the partition. Intuitively, the Gini impurity is the probability of incorrectly classifying a randomly chosen element in the dataset if it were randomly labeled according to the class distribution in the dataset. Other popular impurity measures include (cross-)entropy (also known as information gain) and Bernoulli variance.

Regularization is an important consideration with tree-based models as one can always learn a tree that assigns exactly one training data point to each leaf node, memorizing the training dataset exactly. Regularization methods include restricting the tree building process, based on

- *minimum size*: stop splitting if the resulting children are smaller than a minimum size;
- *maximum depth*: stop splitting if the the resulting children are beyond some maximum tree depth;
- *maximum number of nodes*: stop splitting if the tree already has maximum number of allowable nodes; and
- *minimum reduction in impurity*: stop splitting if resulting children do not reduce impurity by at least $\delta\%$.

Individual trees are known as *weak learners* because they generally perform only slightly better than random guessing. Multiple trees can be combined in various ways via *ensemble methods* to create stronger classifiers. The two main types of tree ensemble methods are *bootstrap aggregation (bagging)* [17], which aims to reduce the variance of low-bias models, and *boosting* [48], which aims to reduce the bias of many low-variance models. The differences between the two methods are illustrated in Fig. 5.20.

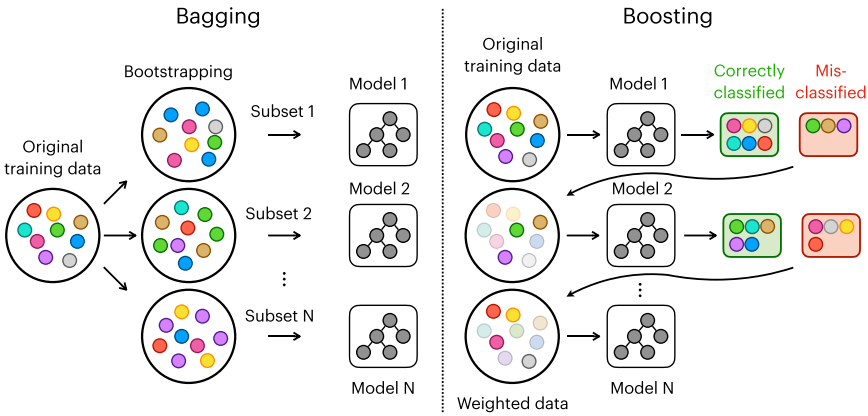


Figure 5.20. Comparison between bagging (left) and boosting (right) ensemble methods for decision trees. In bagging, $N$ models are trained (potentially in parallel) after randomly sampling $N$ subsets from the original training data with replacement. In boosting, $N$ models are trained sequentially by placing higher weight on those events that are misclassified by previous models.

In bagging, the goal is to learn $T$ models and then average the prediction for regression tasks

$$f(x|\theta) = \frac{1}{T}\sum_{t=1}^{T} f_t(x|\theta_t) \tag{5.31}$$

or return the class selected by most trees for classification tasks. Typically, the $T$ training datasets $B_1, \ldots, B_T$, each of size $N$, are resampled with replacement from the original training dataset $S$ (bootstrap resampling). If the $T$ training datasets were completely independent, then the bias of the average model would be the same as the original model, but the variance would be reduced by a factor of $T$. With bootstrap resampling, the bias may increase, but reducing the variance often improves performance.

Random forests [63] combine bagging with the selection of random subsets of attributes. Instead of choosing the best split among all attributes, the best split among a random subset of $k$ attributes is chosen. Random forests are more resistant to overfitting their training set.

One of the first boosting algorithms, adaptive boosting (AdaBoost) [47], builds a sequence of trees $f_1, \ldots, f_T$, each trained with reweighted versions of the original training dataset. The weight of an individual training sample is based on the prediction error of the previous iteration. The loss function

and training procedure for each iteration are modified to account for the weighted training dataset $\{x_i, y_i, w_i\}, i = 1, \ldots, N$.

The procedure is initiated by setting uniform weights $w^{(t=0)} = 1/N$. For classification, the weighted error of the $t$th model is

$$E_t = \frac{\sum_{i=1}^{N} w_i^{(t)} \delta[y_i \neq f_t(x_i|\theta_t)]}{\sum_{i=1}^{N} w_i^{(t)}} \tag{5.32}$$

For highly accurate models, this error is small, $E_t \sim 0$, while for highly inaccurate models, this error may be large, e.g., $E_t \sim 0.5$. Unlike in Eq. (5.31), where the weight of each model is 1, we set a different weight $\beta_t$ for each model depending on the error $\beta_t = \ln[(1-E_t)/E_t]$. For the ensemble prediction, we return the class selected by the trees with the largest sum of weights. Since $\beta_t$ is larger for more accurate models, we prioritize those in the ensemble prediction.

At each iteration, the weights of the misclassified events are updated as $w^{(t+1)} = w^{(t)} \exp(\beta_t)$ and then normalized so that the sum of all the weights is 1. This reweighted dataset is then used to train the next model $f_{t+1}(x|\theta_{t+1})$. As an example, a mediocre classifier, with a misclassification rate $E_t = 30\%$, would have a corresponding $\beta_t = \ln[(1 - 0.3)/0.3)] = 0.85$. So, misclassified events get their weights multiplied by $\exp(0.85) = 2.3$, and the next tree will consider these events to be about twice as important. Now, consider an excellent classifier with an error rate $E_t = 1\%$ and $\beta_t = \ln[(1 - 0.01)/0.01)] = 4.6$. Misclassified events have their influence boosted by a factor of $\exp(4.6) = 99.5$ and thus contribute significantly to the next tree.

In HEP, a popular framework for training BDTs is the Toolkit for Multivariate Data Analysis (TMVA) [65]. More recently, XGBoost [25], which implements a variant of *gradient boosting* [48], has found widespread use in HEP due to its speed, support for GPU acceleration, and integration with the scientific Python ecosystem. Models built with XGBoost have been successfully applied in many HEP data analyses, including winning first place in the Higgs Boson Machine Learning Challenge, hosted on Kaggle [3].

### 5.3.2 *Neural networks*

A feedforward, artificial neural network, also referred to as a multi-layer perceptron, is a collection of units organized into $L$ layers $f = f_L \circ \cdots \circ f_1$. The $\ell$th layer is a mapping from $d_{\ell-1}$ real-valued inputs to $d_\ell$ real-valued outputs, $f_\ell$: $\mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}$. Each layer is implemented as an affine

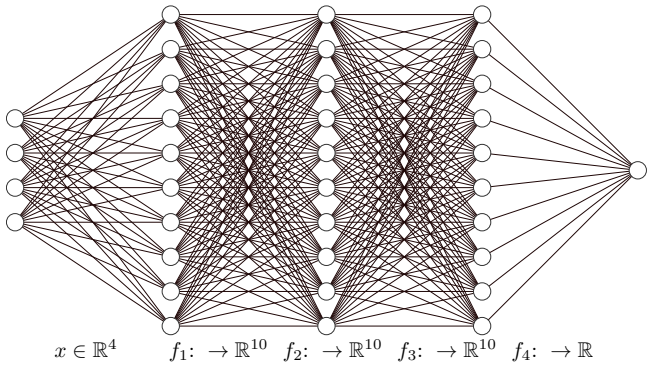


Figure 5.21. Example of a neural network with four layers.

transformation — a multiplication of the input vector $u \in \mathbb{R}^{d_{\ell-1}}$ by a *weight matrix* $W \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ and the addition of a bias vector $b \in \mathbb{R}^{d_\ell}$—together with a pointwise nonlinear *activation function* $\sigma$:

$$f_\ell(u) = \sigma(W_\ell u + b_\ell) \tag{5.33}$$

The purpose of the activation function is to enable learning more complex functions of the input. Without these nonlinearities, the network would be equivalent to a linear model. The parameters of the neural network are the complete set of weights and biases for each layer $\theta = (W_1, \ldots, W_L, b_1, \ldots, b_L)$. An example of a four-layer neural network is shown in Fig. 5.21.

Traditionally, biologically inspired *saturating* activation functions have been used, including the sigmoid function $\text{sigmoid}(u) = 1/(1 + e^{-u})$ and the hyperbolic tangent function $\tanh(u) = (e^u - e^{-u})/(e^u + e^{-u})$. Far from zero input, both sigmoid and tanh saturate at nearly constant values. This can create a problem for gradient-based optimization, especially if the inputs, weights, and biases are not properly scaled so that they take on large positive or negative values. This is known as the "vanishing gradient problem." A popular activation function that partially circumvents this issue is the rectified linear unit (ReLU) [51, 99], $\text{ReLU}(u) = \max(u, 0)$, which is widely used in deep neural networks [59]. However, ReLU suffers a similar saturation problem for negative inputs, known as the "dying ReLU problem," so a variety of alternative solutions have been proposed, including leaky ReLU [91], parameterized ReLU (PReLU) [58], exponential linear unit

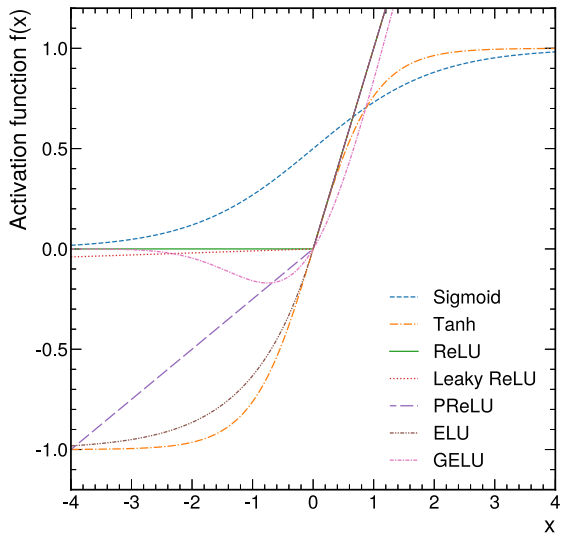


Figure 5.22. Activation functions, including biologically inspired saturating ones, such as sigmoid and tanh, and non-saturating ones, such as ReLU, leaky ReLU, PReLU, ELU, and GELU.

(ELU) [27], and Gaussian error linear unit (GELU) [60]. Visualizations of these different kinds of activation functions are shown in Fig. 5.22.

A softmax function is often used to normalize elements of a discrete vector $u$, or to interpret the output as a probability over a set of $n_C$ discrete categories as in multi-classification tasks. Given a real-valued input vector $u \in \mathbb{R}^{n_C}$, the softmax function computes an output vector $v \in \mathbb{R}^{n_C}$, whose $i$th component is given by

$$\text{softmax}(u)_i = v_i = \frac{\exp(u_i)}{\sum_{j=1}^{n_C} \exp(u_j)} \tag{5.34}$$

The output has the property that $v_i \in (0, 1)$ and $\sum_i v_i = 1$. The input vector components $u_i$ are often referred to as logits, and the softmax function is commonly used as the last layer in multi-class classifier because it is compatible with the CCE loss.

### 5.3.2.1 *Backpropagation*

To train neural networks with gradient descent, we must compute the gradient of the loss function with respect to each parameter. Naively, this

requires many individual computations, but by organizing these computations in a specific way and reusing the outputs of previous computations, we can efficiently compute all of the needed gradients. This is known as the *backpropagation* [111] algorithm, and its basis is the chain rule of calculus.

As an explicit example, consider a two-layer neural network. For simplicity, we ignore the bias parameters. It is a composite function, where we can perform the computations layer by layer. Then, to compute the loss function, we only need the output of the neural network and the target $y$. Writing out these steps explicitly, given an input $x$,

$$z_1 = W_1 x \tag{5.35}$$

$$u_1 = \sigma(z_1) \tag{5.36}$$

$$z_2 = W_2 u_1 \tag{5.37}$$

$$u_2 = \sigma(z_2) \tag{5.38}$$

$$l = L(y, u_2) \tag{5.39}$$

where $W_1$ ($W_2$) is the weight matrix of the first (second) layer, $z_1$ ($z_2$) is the pre-activation output of the first (second) layer, $\sigma$ is the activation function, $u_1$ ($u_2$) is the post-activation output of the first (second) layer, and $l$ is the loss function value. These computational steps are called the "forward pass" because we progress through the network in the forward direction.

To compute the gradient of the loss function with respect to all parameters, it is natural to begin from the last layer. So, let us compute the gradient with respect to $W_2$ in the second layer, denoted $\partial l/\partial W_2$. To do this, we can apply the chain rule to decompose the gradient into three terms:

$$\frac{\partial l}{\partial W_2} = \left(\frac{\partial l}{\partial u_2}\right)\left(\frac{\partial u_2}{\partial z_2}\right)\left(\frac{\partial z_2}{\partial W_2}\right) \tag{5.40}$$

The term $\partial u_2/\partial z_2$ is just the derivative of the nonlinear activation function, which is often easy to compute. We can save the numerical values for each of these separate terms.

Working backward through the network, we can proceed to compute the gradient with respect to $W_1$ in the first layer with the chain rule:

$$\frac{\partial l}{\partial W_1} = \left(\frac{\partial l}{\partial u_2}\right)\left(\frac{\partial u_2}{\partial z_2}\right)\left(\frac{\partial z_2}{\partial u_1}\right)\left(\frac{\partial u_1}{\partial z_1}\right)\left(\frac{\partial z_1}{\partial W_1}\right) \tag{5.41}$$

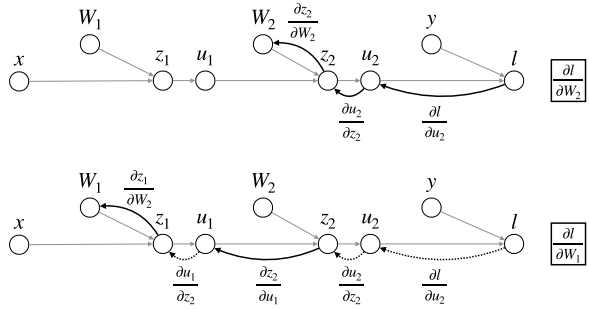


Figure 5.23. Visualizations of the backpropagation algorithm for the computation of $\frac{\partial l}{\partial W_2}$ (upper) and $\frac{\partial l}{\partial W_1}$ (lower). The forward pass is shown with straight gray lines, while the backward pass is shown with curved black lines. For the second computation of $\frac{\partial l}{\partial W_1}$, the reused computations are shown with dotted lines.

Of the five terms, two of them (highlighted in gray) have already been computed in Eq. (5.40). Furthermore, one of the remaining terms is the derivative of the first activation function with respect to its input $\partial u_1/\partial z_1$, which is equal to the previously calculated $\partial u_2/\partial z_2$. So, in order to find the gradient with respect to $W_1$, we only need to perform two additional computations. This is the essence of the backpropagation algorithm. This process of computing and multiplying gradients is known as the "backward pass." These rules can be extended to arbitrarily deep neural networks, as long as each layer and the loss function are differentiable.

Figure 5.23 shows the computational graph of the network, highlighting the computations needed for $\frac{\partial l}{\partial W_2}$ (upper) and $\frac{\partial l}{\partial W_1}$ (lower). Each node is an input, output, or parameter. The forward pass is shown with straight gray lines, while the backward pass is shown with curved black lines. For the second computation of $\frac{\partial l}{\partial W_1}$, the reused computations are shown with dotted lines.

Modern ML software packages implement automatic differentiation (AD), exploiting the fact that neural networks consist of a sequence of elementary arithmetic operations and functions with known derivatives and repeatedly applying the chain rule to compute the target partial derivative automatically.

### 5.3.3 *Convolutional neural networks*

An *inductive bias* expresses assumptions about the data-generating process or the space of solutions, allowing a learning algorithm to prioritize one solution over another [94]. Incorporating an inductive bias into an ML

algorithm can enable them to learn more efficiently, for example, with less data or fewer parameters. These models may also generalize better to unseen data.

For image-like data, there are inductive biases that help carry out common tasks, such as classification, regression, and segmentation:

- *Locality*: Nearby areas within an image tend to contain stronger patterns.
- *Translation equivariance*: Only relative positions within an image are relevant.

As an example task, consider classifying galaxy morphologies [5, 38], e.g., spiral, elliptical, or lenticular. For this task, the solution should not depend on the location of the galaxy within an image. Moreover, many of the identifying characteristics of different types of galaxies are localized in small patches of an image.

Convolutional neural networks (CNNs), as shown in Fig. 5.24, incorporate these inductive biases through their design. An input image is described by a tensor $x \in \mathbb{R}^{H \times W \times C}$, where $H$ is the height of the image in pixels and $W$ is the width of the image in pixels, and at each pixel location, there is a vector of $C$ features or *channels*. For natural images, there are typically three channels representing the red, green, and blue color channels. CNNs implement a convolution of the input image and a *filter*, or *kernel*, with height $J$ and width $K$. The parameters of the filter are learnable and the convolution involves traversing over input and calculating the product of the filter $W$ with a patch of the input, which has the same spatial shape as the filter and is centered at the target pixel. In practice, $M$ filters are combined into a single tensor $W \in \mathbb{R}^{J \times K \times M}$.

We can calculate one element of the output tensor $y \in \mathbb{R}^{V \times U \times M}$ from the input tensor $x$, filter tensor $W$, and length-$M$ bias vector $b$ as[3]

$$y[v, u, m] = \left( \sum_{c=1}^{C} \sum_{j=1}^{J} \sum_{k=1}^{K} W[j, k, c, m]\, x[v + j, u + k, c] \right) + b[m] \quad (5.42)$$

For simplicity, we typically assume $J = K$ (square kernel). By repeating the operation over all the input pixels, the result of a kernel convolution is also an image.

[3]Note that in practice $x$ is shifted by, e.g., $\left(\frac{J+1}{2}, \frac{K+1}{2}\right)$ in order to be symmetric around $(v, u)$.

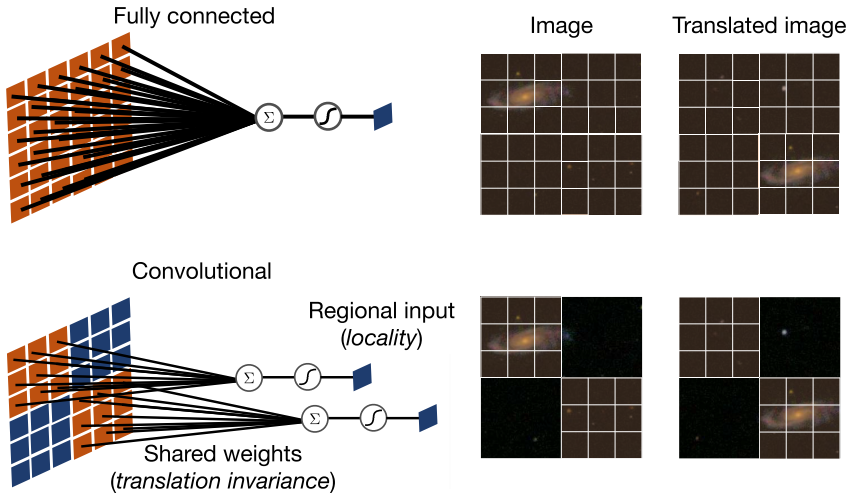


Figure 5.24. Convolutional neural networks incorporate the inductive biases of locality and translation equivariance through their design. A CNN can be interpreted as an MLP with shared weights.

A key feature of convolutions is that they are equivariant to translations: if the input image is shifted $x[i, \dots] \to x[i - j, \dots]$, then the output is also shifted by the same amount $y[v, \dots] \to y[v - j, \dots]$. Another way of looking at this is to compare this to a fully connected MLP. A fully connected MLP acting on the same image as a fully unrolled vector would generally not have this symmetry. Another way of interpreting a CNN is as a very specific type of MLP with shared weights. CNNs generally have fewer parameters than the corresponding fully connected MLP, which can improve the optimization process. The CNN structure allows for patterns in one part of an image in the training dataset effectively contribute to learning that pattern anywhere in the image.

A kernel convolution involves three hyperparameters: the kernel size (typically an odd number so that the filter has an unambiguous center), stride, and padding. In practice, kernel sizes of $1 \times 1$, $3 \times 3$, or $5 \times 5$ are frequently used. A $1 \times 1$ convolution cannot capture correlations among different pixels, but it can increase or decrease the number of features per pixel [59, 86, 119]. The stride is the number of pixels between each target pixel. For example, for a stride of 1, the target pixels are adjacent, whereas for a stride of 2, 1 pixel is skipped along each axis. Padding expands the

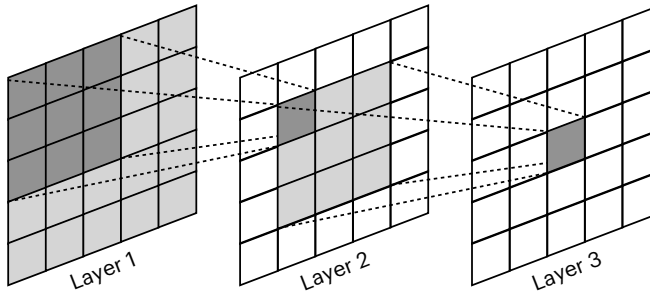


Figure 5.25. Illustration of receptive field in CNNs. Stacking two $3 \times 3$ kernels will lead to a larger receptive field equivalent to a $5 \times 5$ kernel.

input image by a specified number of pixels for when the target pixel is near the edge and the filter would extend beyond the input image.

CNNs can identify features with a spatial size larger than the kernel size by stacking multiple convolutional layers. For example, stacking two $3 \times 3$ kernels will lead to a larger *receptive field* equivalent to a $5 \times 5$ kernel, as shown in Fig. 5.25. Another approach, known as an inception module, extracts features using kernels of different sizes simultaneously [119, 120].

CNNs often use *pooling* to downsample the image, further extending the receptive field. A pooling operation is a type of aggregation that takes many input pixels and produces one output pixel. The most popular pooling operations are max pooling and average pooling. Max pooling picks the highest activation pixel value within the specified receptive field, while average pooling computes the average pixel value in the receptive field. Average pooling tends to smooth out an image, so sharp features may not be preserved. However, a drawback of max pooling is that all pixels other than the maximum one are ignored. Examples of the two operations are shown in Fig. 5.26.

By globally pooling over the entire image, a single feature vector with no spatial index can be created, giving rise to a potentially translation-invariant CNN. Reducing the image size, either through pooling or a convolution with stride larger than 1, can also be computationally beneficial. The reduction in the spatial size of an image is carried out gradually, typically by a factor of 2. After the spatial size is reduced, the number of channels is typically increased (usually by the same factor of 2). CNNs can consist of dozens or sometimes hundreds of convolutional layers, and their

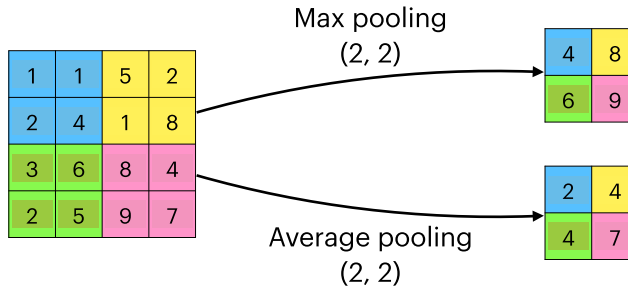


Figure 5.26. Max (upper) and average (lower) pooling in CNNs. Max pooling picks the highest activation pixel value within the specified receptive field, while average pooling computes the average pixel value in the receptive field.

optimization may be challenging due to the vanishing gradient problem. Techniques such as batch normalization [73], which normalize the tensors at each convolution layer, and skip connections [59] can mitigate this and have contributed to the tremendous success of CNNs for image-based tasks.

## 5.4 Applications

Machine learning has found numerous natural applications in analysis reconstruction in particle physics. At the lowest level, machine learning tools can perform hit reconstruction or track finding in individual detector systems. These tools can also identify objects such as electrons, photons, $\tau$ leptons, and jets, using information from various detector systems. Recently, researchers have also explored the use of ML to accelerate or replace computationally intensive detector simulation [4,19]. Finally, ML tools have been widely used to classify entire events as background- or signal-like, both in the final statistical analysis and at the initial trigger decision.

ML tools have found high-profile applications in particle physics. For example, BDTs were instrumental in the discovery of the Higgs boson [1, 23, 24], including in the CMS $H \to \gamma\gamma$, CMS $VH \to b\overline{b}$, and ATLAS $H \to \tau\tau$ analyses. For five key Higgs boson analyses, ML greatly increased the sensitivity of the LHC experiments, improving the discovery $p$-values by factors ranging from about 2–20, or equivalently, reducing the amount of data that would need to be collected by about 13–56% [106].

In this section, we discuss two representative use cases of ML in HEP, intended to highlight unique aspects of HEP data and requirements: jet tagging and trigger applications.

### 5.4.1 *Jet tagging*

Quarks and gluons originating from high energy particle collisions, such as the proton–proton collisions at the LHC, generate a cascade of other particles (mainly other quarks or gluons) that then arrange themselves into hadrons. The stable and unstable hadrons' decay products are observed by large particle detectors, reconstructed by algorithms that combine the information from different detector components and then clustered into *jets*, using physics-motivated sequential recombination algorithms [20–22, 39]. Jet identification, or *tagging*, algorithms are designed to identify the nature of the particle that initiated a given cascade, inferring it from the collective features of the particles generated in the cascade. This is illustrated in Fig. 5.27.

Traditionally, jet tagging was meant to distinguish three classes of jets: light flavor quarks, gluons, or bottom quarks. At the LHC, due to the large collision energy, new jet topologies emerge when heavy particles, e.g., $W$, $Z$, or Higgs bosons or top quarks, are produced with large momentum and decay to all-quark final states. In this case, the resulting jets contain

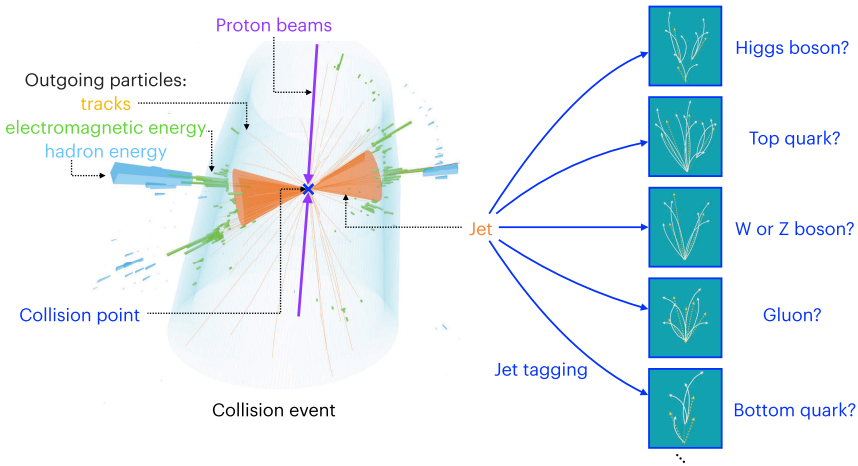


Figure 5.27. A visual representation of a collision event at the LHC and the task of jet tagging. Proton beams (purple arrows) cross at a collision point (blue cross). Outgoing particles make tracks (curved orange lines), energy deposits in the electromagnetic calorimeter (green boxes), and energy deposits in the hadron calorimeter (blue boxes). The orange cone represents a cluster of tracks and energy deposits reconstructed as a jet. The task of jet tagging is to infer, on a statistical basis, the origin of a jet based on its measured characteristics.

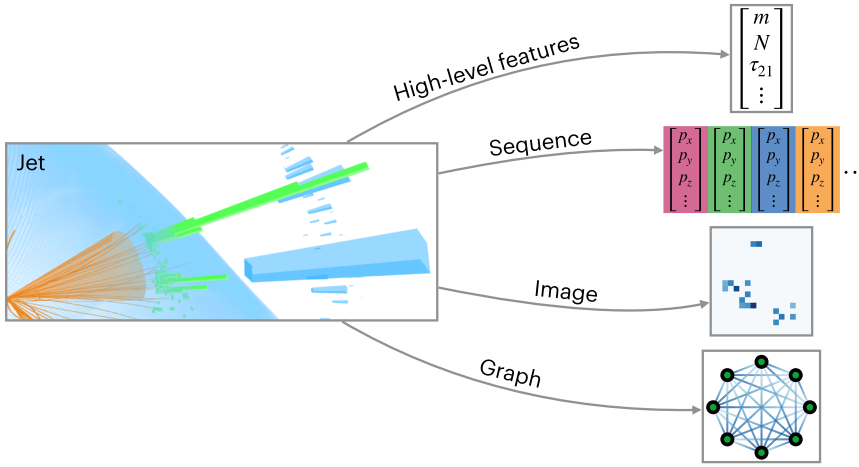


Figure 5.28. Visualization of different jet representations, including high-level features, sequences, images, and graphs.

the overlapping showers of these decay products and can appear similar to showers from single quarks or gluons. These jets are characterized by a large invariant mass and differ from quark and gluon jets in their energy correlations. Several techniques have been proposed to identify these jets by using physics-motivated quantities, collectively referred to as "jet substructure" variables [80].

Machine learning approaches for jet tagging have been extensively investigated using different *representations* of the jet, i.e., ways to encode and preprocess the information, as shown in Fig. 5.28. Different representations are naturally coupled to different kinds of ML models. For example, physics-motivated quantities, also known as high-level features, such as mass, particle multiplicity, or $N$-subjettiness [121] can be processed with fully connected neural networks or BDTs. A lower-level representation consists of treating the particle features as a sequence, list, or set of inputs. This type of representation can be processed by recurrent neural networks (RNNs) [90], which act on each element in a sequence and contain an internal memory, or deep sets [78], whose output is invariant under permutations of the inputs.

Jets can also be preprocessed into two-dimensional images in the $(\eta, \phi)$ plane, in which each pixel value represents the sum of the particle transverse momenta $p_{\mathrm{T}}$ or energies deposited in a given spatial detector cell. Unlike natural images, jet images are typically sparse, with only a small fraction of non-zero pixels. Jet images can be processed by CNNs, albeit potentially

with some modifications, such as larger kernel sizes [32] or specialized layers optimized for sparse inputs [40].

Finally, jets can also be represented as graphs, with nodes representing particles and edges representing pairwise relationships between particles. This graph data can be processed by graph neural networks (GNNs), a class of models for reasoning about explicitly structured data [18, 50, 77, 85, 112, 113, 124]. GNNs have been successfully applied to identify Higgs bosons decaying to bottom quarks and several other types of jets [96, 97, 105]. It is also possible to encode symmetries, such as Lorentz symmetry, or other physics-inspired inductive biases in GNN models [10–13, 52].

### 5.4.2 *Trigger applications*

In HEP, a significant amount of data processing, including data compression, filtering, and selection, takes place in real time even before the data is written to disk. For example, at the LHC, proton–proton collisions occur at a rate of 40 MHz, but only roughly 1 kHz of this can be saved for offline analysis. Out of this factor of 40 000 rejection, a factor of 400 must occur within a few microseconds of the collision, and the remaining factor of 100 must occur in the next ∼100 ms. In addition, resources are often limited and some applications use specialized hardware such as field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs). Developing ML algorithms for low-latency and resource-constrained environments requires specialized techniques.

FPGAs and ASICs are designed for fast parallel processing with low power usage. The most significant difference between FPGAs and ASICs is that FPGAs can be reprogrammed, while ASICs cannot be changed once manufactured. Therefore, FPGA designs are more flexible, typically consume more power, and have slightly larger latencies than the equivalent ASIC designs. ASICs can also be designed to tolerate high levels of radiation through methods like triplication.

FPGAs contain building blocks of logic gates which can be used to construct algorithms by programming the interconnects between the components. The primary building blocks are dedicated arithmetic units or digital signal processors (DSPs), lookup tables (LUTs) for implementing logic, and two different units for storing information: registers or flip-flops (FFs) and block random-access memory (BRAM). FPGAs also contain a large number of input–output (I/O) links to receive input data and transmit output data. A schematic of a generic FPGA is shown in Fig 5.29. Unlike

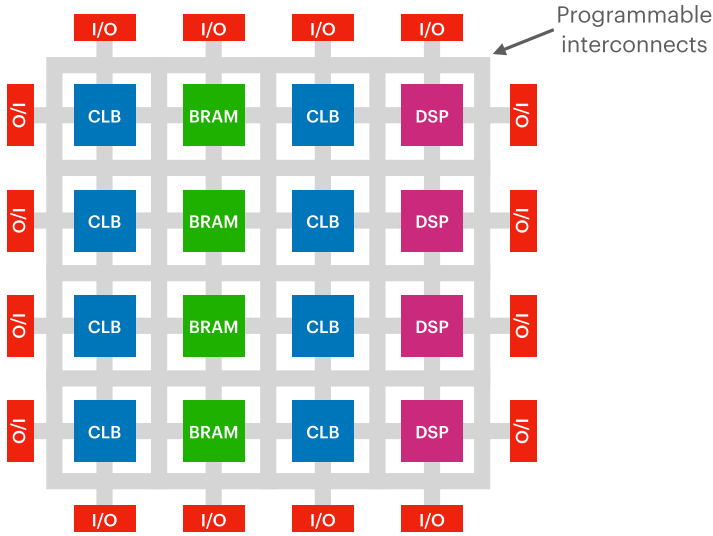


Figure 5.29. Schematic of a generic FPGA. The primary building blocks are digital signal processors (DSPs), lookup tables (LUTs), flip-flops (FFs), block random-access memory (BRAM), and input–output (I/O) links. FFs and LUTs are combined into configurable logic blocks (CLBs).

traditional CPUs, these devices are only capable of running the algorithm(s) that have been programmed. As a result of this specialization and their high clock frequencies (typically hundreds of MHz), algorithms can be executed in $\mathcal{O}$(ns).

Programming FPGAs requires the use of dedicated hardware description languages (HDLs) such as VHDL or Verilog as well as a design methodology that is aware of the limitations and nature of the relevant device. All components of an FPGA program must be synchronized with the rising and falling edges of the clock, and the relations between components must be thought of in relation to these clock periods. Recently, high-level synthesis (HLS) tools [72, 115, 125], which take algorithms written in untimed (typically C) code decorated with directives and produce equivalent HDL algorithms, have lowered the barrier to entry for using FPGAs and ASICs.

Several tools, including hls4ml [43], FINN [9, 122], Conifer [117], and fwXmachina [68], have been developed to automatically create firmware from ML algorithms, as shown in Fig. 5.30. These tools have been used for applications ranging from jet tagging [75] to muon $p_{\mathrm{T}}$ regression [28],

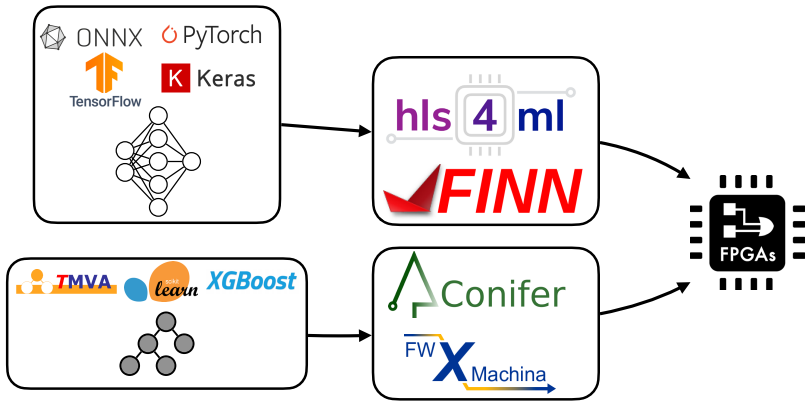


Figure 5.30. Tools like hls4ml [43], FINN [9, 122], Conifer [117], and fwXmachina [68] can translate ML algorithms from libraries like TensorFlow, Keras, PyTorch, ONNX, Scikit-learn, TMVA, or XGBoost into firmware for FPGAs.

on-detector data compression [37], charged particle tracking [45, 69], calorimeter reconstruction [71], and anomaly detection [54]. Hardware-AI co-design principles, including pruning [57], quantization [29], and parallelization [43], are important to consider to produce optimal designs that satisfy strict latency and resource constraints.

## 5.5 Summary and Outlook

Machine learning (ML) is now an integral part of research in high energy physics (HEP), from analysis to instrumentation, reconstruction, and simulation. Beyond being an essential tool, computational methods like ML are a third fundamental approach for studying physics on the same logical level as theory and experiment. In this chapter, we gave an overview of ML basics, types of models, and advanced techniques like model compression and surveyed some recent ML applications in HEP. There are, of course, a plethora of techniques and tools that we could not cover, many of which can be found in the HEP ML Living Review [61].

The ML in HEP research community benefits tremendously from the proliferation of public datasets and research code on GitHub, open-source software packages, such as TensorFlow [2], Keras [26], PyTorch [102], and JAX [15], commercial hardware for ML training and inference, such as NVIDIA GPUs, and widely available learning resources. This culture of openness advances the fast-paced nature of ML development, in which

the latest state-of-the-art methods can be quickly extended and even superseded within months.

Strides in ML and HEP research benefit each other. In one direction, ML has helped revolutionize HEP research by enabling discoveries with less data, model-agnostic searches for exotic new physics, and exploration of final states previously thought impossible. In the other direction, HEP has unique characteristics and challenges, such as the petabyte-scale datasets, enormous data throughput, strict latency and resource constraints, and the physics and symmetry structures underpinning the data, that drive innovation in ML. Despite these benefits, there are valid criticisms of using ML in HEP research, such as the possibility of bias, the need for careful validation and calibration of ML models in data, and the difficulty of reinterpretation of HEP results that heavily rely on ML models.

Beyond the dizzying array of existing HEP applications, there continue to be many new opportunities to apply ML in surprising ways. If advances continue at the current pace, the future is bright for this (still) growing subfield.

## References

[1] G. Aad, *et al.* Observation of a new particle in the search for the standard model Higgs Boson with the ATLAS detector at the LHC. *Physics Letters B*, 716:1, 2012. DOI: 10.1016/j.physletb.2012.08.020, arXiv:1207.7214 [hep-ex].

[2] M. Abadi, *et al.* (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. 1603.04467 [cs.DC], https://www.tensorflow.org/.

[3] C. Adam-Bourdarios, G. Cowan, C. Germain-Renaud, I. Guyon, B. Kégl, and D. Rousseau. The Higgs machine learning challenge. *Journal of Physics: Conference Series*, 664(7):072015, 2015. DOI: 10.1088/1742-6596/664/7/072015.

[4] Y. Alanazi, N. Sato, P. Ambrozewicz, A. N. H. Blin, W. Melnitchouk, M. Battaglieri, T. Liu, and Y. Li. A survey of machine learning-based physics event generation. In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 2021, p. 4286. DOI: 10.24963/ijcai.2021/588, arXiv:2106.00643.

[5] AstroDave, AstroTom, C. R., Winton, joycenv, and K. Willett. Galaxy zoo - the galaxy challenge, 2013. https://kaggle.com/competitions/galaxy-zoo-the-galaxy-challenge.

[6] A. Aurisano, A. Radovic, D. Rocco, A. Himmel, M. D. Messier, E. Niner, G. Pawloski, F. Psihas, A. Sousa, and P. Vahle. A convolutional

neural network neutrino event classifier. *JINST*, 11(9):P09001, 2016. DOI: 10.1088/1748-0221/11/09/P09001, arXiv:1604.01444 [hep-ex].

[7] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry. Post-training 4-bit quantization of convolution networks for rapid-deployment. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc., 2019, p. 7950. arXiv:1810.05723 [cs.CV], https://proceedings.neurips.cc/paper/2019/file/c0a62e133894cdce435bcb4a5df1db2d-Paper.pdf.

[8] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag. What is the state of neural network pruning? In: I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), *Proceedings of Machine Learning and Systems*, Vol. 2, 2020, p. 129. arXiv:2003.03033 [cs.LG], https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

[9] M. Blott, T. Preusser, N. Fraser, G. Gambardella, K. O'Brien, and Y. Umuroglu. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems*, 11:3, 2018. DOI: 10.1145/3242897, arXiv:1809.04570 [cs.AR].

[10] A. Bogatskiy, B. Anderson, J. T. Offermann, M. Roussi, D. W. Miller, and R. Kondor. Lorentz group equivariant neural network for particle physics. In: H. Daume III and A. Singh (Eds.), *Proceedings of the 37th International Conference on Machine Learning*, Vol. 119, 2020, p. 992. arXiv:2006.04780 [hep-ph], https://proceedings.mlr.press/v119/bogatskiy20a.html.

[11] A. Bogatskiy, T. Hoffman, D. W. Miller, and J. T. Offermann. PELICAN: Permutation equivariant and Lorentz invariant or covariant aggregator network for particle physics. In: *Machine Learning and the Physical Sciences Workshop at Neural Informance Processing Systems*, 2022. arXiv:2211.00454 [hep-ph], https://ml4physicalsciences.github.io/2022/files/NeurIPS_ML4PS_2022_132.pdf.

[12] A. Bogatskiy, T. Hoffman, D. W. Miller, J. T. Offermann, and X. Liu. Explainable equivariant neural networks for particle physics: PELICAN, 2023. arXiv:2307.16506 [hep-ph].

[13] Bogatskiy, A. *et al.* Symmetry group equivariant architectures for physics. In: *Snowmass 2021*, 2022. arXiv:2203.06153 [cs.LG].

[14] D. Bourilkov. Machine and deep learning applications in particle physics. *International Journal of Modern Physics A*, 34:1930019, 2020. DOI: 10.1142/S0217751X19300199, arXiv:1912.08245 [physics.data-an].

[15] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: Composable transformations of Python+NumPy programs, 2018. http://github.com/google/jax.

[16] L. Breiman. *Classification and Regression Trees*, 1st edn. Routledge, 1984. DOI: 10.1201/9781315139470.

[17] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123, 1996. DOI: 10.1007/BF00058655.

[18] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34:18, 2017. DOI: 10.1109/MSP.2017.2693418, arXiv:1611.08097 [cs.CV].

[19] A. Butter and T. Plehn. Generative networks for LHC events. In: P. Calafiura, D. Rousseau, and K. Terao (Eds.), *Artificial Intelligence for High Energy Physics*. World Scientific, 2022, p. 191. DOI: 10.1142/9789811234033_0007, arXiv:2008.08558.

[20] M. Cacciari, G. P. Salam, and G. Soyez. The anti-$k_t$ jet clustering algorithm. *JHEP*, 04:063, 2008. DOI: 10.1088/1126-6708/2008/04/063, arXiv:0802.1189 [hep-ph].

[21] M. Cacciari, G. P. Salam, and G. Soyez. FastJet user manual. *The European Physical Journal C*, 72:1896, 2012. DOI: 10.1140/epjc/s10052-012-1896-2, arXiv:1111.6097 [hep-ph].

[22] S. Catani, Y. L. Dokshitzer, M. H. Seymour, and B. R. Webber. Longitudinally invariant $K_t$ clustering algorithms for hadron hadron collisions. *Nuclear Physics B*, 406:187–224, 1993. DOI: 10.1016/0550-3213(93)90166-M.

[23] S. Chatrchyan, *et al.* Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Physics Letters B*, 716:30, 2012. DOI: 10.1016/j.physletb.2012.08.021, arXiv:1207.7235 [hep-ex].

[24] S. Chatrchyan, *et al.* Observation of a new boson with mass near 125 GeV in $pp$ collisions at $\sqrt{s} = 7$ and 8 TeV. *JHEP*, 06:81, 2013. DOI: 10.1007/JHEP06(2013)081, arXiv:1303.4571 [hep-ex].

[25] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, New York, 2016, p. 785. DOI: 10.1145/2939672.2939785.

[26] F. Chollet, *et al.* Keras, 2015. https://keras.io.

[27] D. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). In: Y. Bengio and Y. LeCun (Eds.), *4th International Conference on Learning Representations (ICLR), Conference Track Proceedings*, 2016. arXiv:1511.07289 [cs.LG].

[28] CMS Collaboration. The phase-2 upgrade of the CMS Level-1 trigger. CMS Technical Design Report CERN-LHCC-2020-004. CMS-TDR-021, 2020. https://cds.cern.ch/record/2714892.

[29] C. N. Coelho, A. Kuusela, S. Li, H. Zhuang, J. Ngadiuba, T. K. Aarrestad, V. Loncar, M. Pierini, A. A. Pol, and S. Summers. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Machine Intelligence*, 2021. DOI: 10.1038/s42256-021-00356-5, arXiv:2006.10159.

[30] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training deep neural networks with binary weights during propagations. In: C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc., 2015, p. 3123. 1511.00363, https://proceedings.neurips.cc/paper/2015/file/3e15cc11f979ed25912dff5b0669f2cd-Paper.pdf.

[31] I. Csabai, F. Czako, and Z. Fodor. Quark and gluon jet separation using neural networks. *Physical Review D*, 44:1905, 1991. DOI: 10.1103/PhysRevD.44.R1905.

[32] L. de Oliveira, M. Kagan, L. Mackey, B. Nachman, and A. Schwartzman. Jet-images — deep learning edition. *Journal of High Energy Physics*, 7:69, 2016. DOI: 10.1007/JHEP07(2016)069, arXiv:1511.05190 [hep-ph].

[33] B. H. Denby. Neural networks and cellular automata in experimental high-energy physics. *Computer Physics Communications*, 49:429, 1988. DOI: 10.1016/0010-4655(88)90004-5.

[34] B. H. Denby. The use of neural networks in high-energy physics. *Neural Computation*, 5:505, 1993. DOI: 10.1162/neco.1993.5.4.505.

[35] B. H. Denby, T. Lindblad, C. S. Lindsey, G. Szekely, J. Molnar, A. Eide, S. R. Amendolia, and A. Spaziani. Investigation of a VLSI neutral network chip as part of a secondary vertex trigger. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 335:296, 1993. DOI: 10.1016/0168-9002(93)90284-O.

[36] G. Di Guglielmo, *et al.* Compressing deep neural networks on FPGAs to binary and ternary precision with `hls4ml`. *Machine Learning: Science and Technology*, 2(1):015001, 2020. DOI: 10.1088/2632-2153/aba042, 2003.06308.

[37] G. Di Guglielmo, *et al.* A reconfigurable neural network ASIC for detector front-end data compression at the HL-LHC. *IEEE Transactions on Nuclear Science*, 68(8):2179, 2021. DOI: 10.1109/TNS.2021.3087100, arXiv:2105.01683 [physics.ins-det].

[38] S. Dieleman, K. W. Willett, and J. Dambre. Rotation-invariant convolutional neural networks for galaxy morphology prediction. *MNRAS*, 450(2):1441, 2015. DOI: 10.1093/mnras/stv632, arXiv:1503.07077 [astro-ph.IM].

[39] Y. L. Dokshitzer, G. D. Leder, S. Moretti, and B. R. Webber. Better jet clustering algorithms. *JHEP*, 8:1, 1997. DOI: 10.1088/1126-6708/1997/08/001, arXiv:hep-ph/9707323 [hep-ph].

[40] L. Dominé and K. Terao. Scalable deep convolutional neural networks for sparse, locally dense liquid argon time projection chamber data. *Physical Review D*, 102(1):012005, 2020. DOI: 10.1103/PhysRevD.102.012005, arXiv:1903.05663 [hep-ex].

[41] Z. Dong, Z. Yao, Y. Cai, D. Arfeen, A. Gholami, M. W. Mahoney, and K. Keutzer. HAWQ-V2: Hessian aware trace-weighted quantization

of neural networks. In: H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 2020, p. 18518. arXiv:1911.03852 [cs.CV], https://proceedings.neurips.cc/paper/2020/file/d77c703536718b95308130ff2e5cf9ee-Paper.pdf.

[42] Z. Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer. HAWQ: Hessian aware quantization of neural networks with mixed-precision. In: *2019 IEEE/CVF International Conference on Computer Vision*, Seoul, South Korea, October 27, 2019, p. 293. DOI: 10.1109/ICCV.2019.00038, 1905.03696.

[43] J. Duarte, *et al.* Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13:P07027, 2018. DOI: 10.1088/1748-0221/13/07/P07027, arXiv:1804.06913 [physics.ins-det].

[44] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121, 2011. http://jmlr.org/papers/v12/duchi11a.html.

[45] A. Elabd, *et al.* Graph neural networks for charged particle tracking on FPGAs. *Frontiers in Big Data*, 5, 2022. DOI: 10.3389/fdata.2022.828666, arXiv:2112.02048 [physics.ins-det].

[46] J. Frankle and M. Carbin. The lottery ticket hypothesis: Training pruned neural networks. In: *7th International Conference on Learning Representations*, 2019. arXiv:1803.03635, https://openreview.net/forum?id=rJl-b3RcF7.

[47] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119, 1997. DOI: 10.1006/jcss.1997.1504.

[48] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189, 2001. DOI: 10.1214/aos/1013203451.

[49] G. Cowan. Chapter 40: Statistics. In: *Particle Data Group*. R. L. Workman, *et al.* Review of particle physics. *PTEP*, 2022:083C01, 2022. DOI: 10.1093/ptep/ptac097, https://pdg.lbl.gov/2023/reviews/rpp2022-rev-statistics.pdf.

[50] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In: *Proceedings of the 34th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, Vol. 70, 2017, p. 1263. arXiv:1704.01212 [cs.LG], http://proceedings.mlr.press/v70/gilmer17a.html.

[51] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In: G. Gordon, D. Dunson, and M. Dudík (Eds.), *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, Vol. 15, 2011, p. 315. http://proceedings.mlr.press/v15/glorot11a.html.

[52] S. Gong, Q. Meng, J. Zhang, H. Qu, C. Li, S. Qian, W. Du, Z.-M. Ma, and T.-Y. Liu. An efficient Lorentz equivariant graph neural network for jet tagging. *JHEP*, 7:30, 2022. DOI: 10.1007/JHEP07(2022)030, arXiv:2201.08187 [hep-ph].

[53] M. C. Gonzalez-Garcia and M. Yokoyama. Neutrino masses, mixing, and oscillations. In: *Review of Particle Physics*, Vol. 2022, 2022, p. 083C01. DOI: 10.1093/ptep/ptac097, https://pdg.lbl.gov/2023/reviews/rpp2022-rev-neutrino-mixing.pdf.

[54] E. Govorkova, *et al.* Autoencoders on field-programmable gate arrays for real-time, unsupervised new physics detection at 40 MHz at the Large Hadron Collider. *Nature Machine Intelligence*, 4:154–161, 2022. DOI: 10.1038/s42256-022-00441-3, arXiv:2108.03986 [physics.ins-det].

[55] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2018. arXiv:1706.02677 [cs.CV].

[56] D. Guest, K. Cranmer, and D. Whiteson. Deep learning and its application to LHC physics. *Annual Review of Nuclear and Particle Science*, 68: 161, 2018. DOI: 10.1146/annurev-nucl-101917-021019, arXiv:1806.11484 [hep-ex].

[57] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In: Y. Bengio and Y. LeCun (Eds.), *4th International Conference on Learning Representations*, San Juan, Puerto Rico, 2 May 2016. arXiv:1510.00149 [cs.CV].

[58] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, p. 1026. DOI: 10.1109/ICCV.2015.123, 1502.01852.

[59] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[60] D. Hendrycks and K. Gimpel. Gaussian error linear units (GELUs), 2023. arXiv:1606.08415 [cs.LG].

[61] HEP ML Community. A living review of machine learning for particle physics, 2021. arXiv:2102.02770 [hep-ph], https://iml-wg.github.io/HEPML-LivingReview.

[62] G. Hinton. Coursera neural networks for machine learning lecture 6, 2018. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[63] T. K. Ho. Random decision forests. In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 1, 1995, p. 278. DOI: 10.1109/ICDAR.1995.598994.

[64] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735, 1997. DOI: 10.1162/neco.1997.9.8.1735, https://doi.org/10.1162/neco.1997.9.8.1735.

[65] A. Hocker, *et al.* TMVA - Toolkit for multivariate data analysis, 2007. arXiv:physics/0703039.

[66] E. Hoffer, I. Hubara, and D. Soudry. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks, 2018. arXiv:1705.08741 [stat.ML].

[67] T. Hofmann, B. Schölkopf, and A. J. Smola. Kernel methods in machine learning. *Annals of Statistics*, 36(3), 2008. DOI: 10.1214/009053607000000677, math/0701907.

[68] T. M. Hong, B. Carlson, B. Eubanks, S. Racz, S. Roche, J. Stelzer, and D. Stumpp. Nanosecond machine learning event classification with boosted decision trees in FPGA for high energy physics. *JINST*, 16(8):P08016, 2021. DOI: 10.1088/1748-0221/16/08/P08016, arXiv:2104.03408 [hep-ex].

[69] S.-Y. Huang, Y.-C. Yang, Y.-R. Su, B.-C. Lai, J. Duarte, S. Hauck, S.-C. Hsu, J.-X. Hu, and M. S. Neubauer. Low latency edge classification GNN for particle trajectory tracking on FPGAs. In: *33rd International Conference on Field-Programmable Logic and Applications*, 2023. arXiv:2306.11330 [cs.AR].

[70] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1, 2018. 1609.07061, http://jmlr.org/papers/v18/16-456.html.

[71] Y. Iiyama, *et al.* Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction in high energy physics. *Frontiers in Big Data*, 3:44, 2021. DOI: 10.3389/fdata.2020.598927, arXiv:2008.03601 [hep-ex].

[72] Intel. Intel high level synthesis compiler, 2023. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[73] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: F. Bach and D. Blei (Eds.), *32nd International Conference on Machine Learning*, Vol. 37. PMLR, Lille, France, 2015, p. 448. arXiv:1502.03167, http://proceedings.mlr.press/v37/ioffe15.html.

[74] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2017. arXiv:1609.04836 [cs.LG].

[75] E. E. Khoda, *et al.* Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml. *Machine Learning: Science and Technology*, 4(2):025004, 2023. DOI: 10.1088/2632-2153/acc0d7, arXiv:2207.00559 [cs.LG].

[76] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In: Y. Bengio and Y. LeCun (Eds.), *3rd International Conference on Learning Representations (ICLR), Conference Track Proceedings*, 2015. arXiv:1412.6980 [cs.LG].

[77] T. Kipf, E. Fetaya, K.-C. Wang, M. Welling, and R. Zemel. Neural relational inference for interacting systems. In: J. Dy and A. Krause (Eds.), *Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research*, Vol. 80. PMLR, Stockholmsmässan, Stockholm, Sweden, 2018, p. 2688. arXiv:1802.04687 [stat.ML], http://proceedings.mlr.press/v80/kipf18a.html.

[78] P. T. Komiske, E. M. Metodiev, and J. Thaler. Energy flow networks: Deep sets for particle jets. *Journal of High Energy Physics*, 1:121, 2019. DOI: 10.1007/JHEP01(2019)121, arXiv:1810.05165 [hep-ph].

[79] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems — Volume 1*. NIPS'12. Curran Associates Inc., Red Hook, NY, USA, 2012, p. 1097.

[80] A. J. Larkoski, I. Moult, and B. Nachman. Jet substructure at the large hadron collider: A review of recent advances in theory and machine learning. *Physics Reports*, 841:1, 2020. DOI: 10.1016/j.physrep.2019.11.001, arXiv:1709.04464 [hep-ph].

[81] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In: D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems*, Vol. 2. Morgan-Kaufmann, 1990, p. 598. http://papers.nips.cc/paper/250-optimal-brain-damage.

[82] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. *Efficient BackProp*. Springer, Berlin, 2012, pp. 9–48. DOI: 10.1007/978-3-642-35289-8_3.

[83] F. Li and B. Liu. Ternary weight networks, 2016. arXiv:1605.04711.

[84] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-tzur, M. Hardt, B. Recht, and A. Talwalkar. A system for massively parallel hyperparameter tuning. In: I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), *Proceedings of Machine Learning and Systems*, Vol. 2, 2020, p. 230. 1810.05934, https://proceedings.mlsys.org/paper/2020/file/f4b9ec30ad9f68f89b29639786cb62ef-Paper.pdf.

[85] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia. Learning deep generative models of graphs. In: *6th International Conference on Learning Representations, Workshop Track*, 2018. arXiv:1803.03324 [cs.LG], https://openreview.net/forum?id=Hy1d-ebAb.

[86] M. Lin, Q. Chen, and S. Yan. Network in network, 2014. arXiv:1312.4400 [cs.NE].

[87] C. S. Lindsey, B. H. Denby, and H. Haggerty. Drift chamber tracking with neural networks. *IEEE Transactions on Nuclear Science*, 40:607, 1993. DOI: 10.1109/23.256626.

[88] L. Lonnblad, C. Peterson, and T. Rognvaldsson. Finding gluon jets with a neural trigger. *Physical Review Letters*, 65:1321, 1990. DOI: 10.1103/PhysRevLett.65.1321.

[89] L. Lonnblad, C. Peterson, and T. Rognvaldsson. Using neural networks to identify jets. *Nuclear Physics B*, 349:675–702, 1991. DOI: 10.1016/0550-3213(91)90392-B.

[90] G. Louppe, K. Cho, C. Becot, and K. Cranmer. QCD-aware recursive neural networks for jet physics. *Journal of High Energy Physics*, 1:57, 2019. DOI: 10.1007/JHEP01(2019)057, arXiv:1702.00748 [hep-ph].

[91] A. Maas, A. Hannun, and A. Ng. Rectifier nonlinearities improve neural network acoustic models. In: *Proceedings of the International Conference on Machine Learning*, Atlanta, Georgia, 2013.

[92] E. Meller, A. Finkelstein, U. Almog, and M. Grobman. Same, same but different: Recovering neural network quantization error through weight factorization. In: K. Chaudhuri and R. Salakhutdinov (Eds.), *Proceedings of the 36th International Conference on Machine Learning*, Long Beach, CA, USA, Vol. 97. PMLR, 9 June 2019, p. 4486. arXiv:1902.01917 [cs.LG], http://proceedings.mlr.press/v97/meller19a.html.

[93] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. In: *6th International Conference on Learning Representations*, Vancouver, BC, Canada, 30 April 2018. 1710.03740, https://openreview.net/forum?id=r1gs9JgRZ, https://openreview.net/forum?id=r1gs9JgRZ.

[94] T. M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University, New Brunswick, NJ, 1980. http s://www.cs.cmu.edu/~tom/pubs/NeedForBias_1980.pdf.

[95] B. Moons, K. Goetschalckx, N. V. Berckelaer, and M. Verhelst. Minimum energy quantized neural networks. In: M. B. Matthews (Ed.), *2017 51st Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, 29 October 2017, p. 1921. DOI: 10.1109/ACSSC.2017.8335699, arXiv:1711.00215.

[96] E. A. Moreno, *et al.* Interaction networks for the identification of boosted $H \to b\overline{b}$ decays. *Physical Review D*, 102:012010, 2020. DOI: 10.1103/PhysRevD.102.012010, arXiv:1909.12285 [hep-ex].

[97] E. A. Moreno, *et al.* JEDI-net: A jet identification algorithm based on interaction networks. *The European Physical Journal C*, 80:58, 2020. DOI: 10.1140/epjc/s10052-020-7608-4, arXiv:1908.05318 [hep-ex].

[98] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling. Data-free quantization through weight equalization and bias correction. In: *2019 IEEE/CVF International Conference on Computer Vision*, Seoul, South Korea, 27 October 2019, p. 1325. DOI: 10.1109/ICCV.2019.00141, arXiv:1906.04721 [cs.LG].

[99] V. Nair and G. E. Hinton. Rectified linear units improve restricted Boltzmann machines. In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010, p. 807. https://icml.cc/Conferences/2010/papers/432.pdf.

[100] Y. E. Nesterov. A method of solving a convex programming problem with convergence rate $O(\frac{1}{k^2})$. *Doklady Akademii Nauk SSSR*, 269:543, 1983.

[101] R. Ormiston, T. Nguyen, M. Coughlin, R. X. Adhikari, and E. Katsavounidis. Noise reduction in gravitational-wave data via deep learning. *Physical*

*Review Research*, 2(3):033066, 2020. DOI: 10.1103/PhysRevResearch.2.033066, arXiv:2005.06534 [astro-ph.IM].

[102] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc., 2019, p. 8024. arXiv:1912.01703, http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[103] J. Pata, J. Duarte, J.-R. Vlimant, M. Pierini, and M. Spiropulu. MLPF: Efficient machine-learned particle-flow reconstruction using graph neural networks. *The European Physical Journal C*, 81(5):381, 2021. DOI: 10.1140/epjc/s10052-021-09158-w, arXiv:2101.08578 [physics.data-an].

[104] J. Pata, E. Wulff, F. Mokhtar, D. Southwick, M. Zhang, M. Girone, and J. Duarte. Improved particle-flow event reconstruction with scalable neural networks for current and future particle detectors, 2023. arXiv:2309.06782 [physics.data-an].

[105] H. Qu and L. Gouskos. ParticleNet: Jet tagging via particle clouds. *Physical Review D*, 101:056019, 2020. DOI: 10.1103/PhysRevD.101.056019, arXiv:1902.08570 [hep-ph].

[106] A. Radovic, M. Williams, D. Rousseau, M. Kagan, D. Bonacorsi, A. Himmel, A. Aurisano, K. Terao, and T. Wongjirad. Machine learning at the energy and intensity frontiers of particle physics. *Nature*, 560:41, 2018. DOI: 10.1038/s41586-018-0361-2.

[107] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet classification using binary convolutional neural networks. In: *14th European Conference on Computer Vision (ECCV)*. Springer International Publishing, Cham, Switzerland, 2016, p. 525. DOI: 10.1007/978-3-319-46493-0_32, arXiv:1603.05279.

[108] A. Renda, J. Frankle, and M. Carbin. Comparing rewinding and fine-tuning in neural network pruning. In: *8th International Conference on Learning Representations*, Addis Ababa, Ethiopia, 26 April 2020. arXiv:2003.02389 [cs.LG], https://openreview.net/forum?id=S1gSj0NKvB, https://openreview.net/forum?id=S1gSj0NKvB.

[109] B. P. Roe, H.-J. Yang, J. Zhu, Y. Liu, I. Stancu, and G. McGregor. Boosted decision trees as an alternative to artificial neural networks for particle identification. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 543(2):577, 2005. DOI: 10.1016/j.nima.2004.12.018, arXiv:physics/0408124.

[110] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986. DOI: 10.1038/323533a0.

[111] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986. DOI: 10.1038/323533a0.

[112] A. Santoro, D. Raposo, D. G. T. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap. A simple neural network module for relational reasoning. In: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), *Advances in Neural Information Processing Systems 30*, Vol. 30. Curran Associates, Inc., 2017, p. 4967. arXiv:1706.01427 [cs.CL], https://papers.nips.cc/paper/2017/hash/e6acf4b0f69f6f6e60e9a815938aa1ff-Abstract.html.

[113] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks and Learning Systems*, 20(1):61, 2009. DOI: 10.1109/TNN.2008.2005605.

[114] B. Scholkopf and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Adaptive Computation and Machine Learning Series. MIT Press, 2018.

[115] Siemens. Catapult high-level synthesis and verification, 2023. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.

[116] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929, 2014. http://jmlr.org/papers/v15/srivastava14a.html.

[117] S. Summers, *et al.* Fast inference of boosted decision trees in FPGAs for particle physics. *Journal of Instrumentation*, 15:P05026, 2020. DOI: 10.1088/1748-0221/15/05/p05026, arXiv:2002.02534 [physics.comp-ph].

[118] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In: S. Dasgupta and D. McAllester (Eds.), *Proceedings of the 30th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, Vol. 28. PMLR, Atlanta, Georgia, USA, 2013, p. 1139. https://proceedings.mlr.press/v28/sutskever13.html.

[119] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, p. 1. DOI: 10.1109/CVPR.2015.7298594, 1409.4842.

[120] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, p. 2818. DOI: 10.1109/CVPR.2016.308, 1512.00567.

[121] J. Thaler and K. Van Tilburg. Identifying boosted objects with N-subjettiness. *JHEP*, 03:015, 2011. DOI: 10.1007/JHEP03(2011)015, arXiv:1011.2268 [hep-ph].

[122] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A framework for fast, scalable binarized neural network inference. In: *Proceedings of the 2017 ACM/SIGDA International*

*Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, USA, 2017, p. 65. DOI: 10.1145/3020078.3021744, arXiv:1612.07119.

[123] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc., 2018, p. 7675. arXiv:1812.08011 [cs.LG], https://proceedings.neurips.cc/paper/2018/file/335d3d1cd7ef05ec77714a215134914c-Paper.pdf.

[124] X. Wang, R. Girshick, A. Gupta, and K. He. Non-local neural networks. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, p. 7794. DOI: 10.1109/CVPR.2018.00813, arXiv:1711.07971 [cs.CV].

[125] Xilinx. Vitis unified software platform overview, 2023. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[126] Y. You, I. Gitman, and B. Ginsburg. Large batch training of convolutional networks, 2017. arXiv:1708.03888 [cs.CV].

[127] M. D. Zeiler. Adadelta: An adaptive learning rate method, 2012. arXiv:1212.5701 [cs.LG].

[128] D. Zhang, J. Yang, D. Ye, and G. Hua. LQ-nets: Learned quantization for highly accurate and compact deep neural networks. In: V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss (Eds.), *Proceedings of the European Conference on Computer Vision*, Munich, Germany, 8 September 2018, p. 373. DOI: 10.1007/978-3-030-01237-3_23, arXiv:1807.10029 [cs.CV].

[129] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, and Z. Zhang. Improving neural network quantization without retraining using outlier channel splitting. In: K. Chaudhuri and R. Salakhutdinov (Eds.), *Proceedings of the 36th International Conference on Machine Learning*, 9 June 2019, Long Beach, CA, USA, Vol. 97. PMLR, 2019, p. 7543. arXiv:1901.09504 [cs.LG], http://proceedings.mlr.press/v97/zhao19c.html.

[130] H. Zhou, J. Lan, R. Liu, and J. Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc., 2019, p. 3597. arXiv:1905.01067 [cs.LG], https://proceedings.neurips.cc/paper/2019/file/1113d7a76ffceca1bb350bfe145467c6-Paper.

[131] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients, 2016. 1606.06160.

[132] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid. Towards effective low-bitwidth convolutional neural networks. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Salt Lake City, UT, USA, 18 June 2018, p. 7920. DOI: 10.1109/CVPR.2018.00826, arXiv:1711.00205 [cs.CV].