41. Machine Learning

Written August 2025 by J.M. Duarte (UC San Diego), U. Seljak (UC Berkeley; LBNL) and K. Terao (SLAC; Stanford U.).

41.1 Introduction

This chapter gives an overview of the core concepts of machine learning (ML)—the use of algorithms that learn from data, identify patterns, and make predictions or decisions without being explicitly programmed—that are relevant to particle physics with some examples of applications to the energy, intensity, cosmic, and accelerator frontiers. ML is an enormous field that has grown substantially in the last decade, largely driven by the emergence of so-called deep learning (DL) [1, 2]. ML has a long history in particle physics going back to the late 1980s and early 1990s; see Refs. [3–5] for recent reviews. ML is a subset of artificial intelligence (AI), which generally refers to computational systems that can perform tasks typically associated with human intelligence, such as learning, reasoning, problem-solving, perception, and decision-making.

Physicists are exploring and contributing to machine learning at an unprecedented rate, which poses a challenge for those who wish to have an up-to-date view of the field. This motivated an effort to create *A Living Review of Machine Learning for Particle and Nuclear Physics* [6], which can be accessed here: `https://iml-wg.github.io/HEPML-LivingReview/`. At the time of writing, the Living Review included more than 1,800 references organized hierarchically by topic. Although we make references to some of these papers, this chapter focuses on the methodology and does not attempt to give a comprehensive review of the applications.

Machine learning and artificial intelligence have a mathematical foundation that is closely tied to statistics (see Ch. 40), the calculus of variations, approximation theory, and optimal control theory. Nevertheless, there have been tremendous advances in recent years, driven by increased computational power, enormous datasets, and new insights, that are impacting physics and society.

The topic can be organized along a few axes, which we use to structure this section. First, there are different learning paradigms, for example, supervised learning, unsupervised learning, and reinforcement learning. Within these paradigms, there are various tasks; for example, classification and regression—which have been the primary use of ML in particle physics—are examples of supervised learning. In addition to the learning paradigm and tasks, there are various types of machine learning models that generically process some input and produce some output. The types of models vary based on what they are modeling (*e.g.*, so-called discriminative vs. generative models), as well as how they are implemented (*e.g.*, neural networks, decision trees, or kernel machines). Next, there are the issues around training or learning within the context of a given task and model class, which connects to optimization and regularization. We will briefly discuss the various considerations that emerge in the application of machine learning methods to physics, such as the treatment of systematic uncertainty, the interpretability of the models, and the incorporation of symmetry.

41.1.1 *A gentle introduction with a representative example*

We will use a specific, familiar example to introduce the various ingredients in context before factorizing and abstracting them. Consider the task of *classifying* energy deposits in a particle detector as coming from electrons or protons. For this example, let the detector data consist of energy deposits in d sensors so that the data can be represented as a *feature vector* $x \in \mathbb{R}^d$. Different components of x may correspond to physical quantities with different units (*e.g.*, units of energy, momentum, or position).

Due to the complex interactions of particles in the detector, we do not have an explicit probability model for the high-dimensional data for the electron and proton scenarios, but we do have a simulator that allows us to generate Monte Carlo samples for each. This allows us to assemble a *training dataset* $\{x_i, y_i\}_{i=1,\ldots,n}$, where y is a *label* that identifies how the example was generated (*e.g.*, $y = 0$ for electrons and $y = 1$ for protons). We would like to find a function that accurately *predicts* the label on new data. Because we have feature-label pairs, this is a *supervised learning* problem. We can use a *neural network* to provide a flexible family of functions $f_\phi : \mathbb{R}^d \to \mathbb{R}$, where ϕ denotes the internal parameters of the neural network (*i.e.*, the weights and biases that we will discuss in Sec. 41.8.4). The goal of *training* is to find the value of the parameters ϕ that provide the 'best' predictions. This is made concrete through a *loss function* $\mathcal{L}(y, f_\phi(x))$. Instead of the obvious zero-one loss, which is 0 if $f_\phi(x) = y$ and 1 if not, we use the squared-loss $\mathcal{L}_{\text{sq}}(y, f_\phi(x)) = (y - f_\phi(x))^2$, which will be motivated in Sec. 41.2.3. We can evaluate the average of the loss on the training set of size n, known as the *empirical risk* or *training loss* $\mathcal{R}_{\text{emp}}(f_\phi) = \sum_{i=1}^{n} \mathcal{L}(y_i, f_\phi(x_i))/n$. *Training* refers to numerically minimizing the empirical risk. We can numerically optimize the model through *gradient descent*, which iteratively adjusts the parameters of the network according to $\phi^{t+1} = \phi^t - \lambda \nabla_\phi \mathcal{R}_{\text{emp}}(f_\phi)$, where λ is the *learning rate*.

Once the optimization is complete and we obtain the solution $\hat{\phi}$, it is natural to assess the quality of the trained model $f_{\hat{\phi}}$ on an independent *testing dataset*[1]. The empirical risk evaluated on the testing set is often larger than on the training set, and large differences indicate *overfitting*, meaning that the model does not generalize well to the unseen data. The ability to accurately predict on unseen data is referred to as *generalization* and the empirical risk on the test data provides a measure of the *generalization error*. In order to reduce the generalization error one might explore different model choices (*e.g.*, neural network architectures), additional regularization terms in the loss function, different learning rates, optimization algorithms, or early stopping criterion in the optimization.

In order to produce a binary electron vs. proton decision from the continuous output of the neural network, one typically chooses a threshold (*i.e.*, classify as proton if $f_{\hat{\phi}}(x) > c$). The choice of the threshold c is often referred to as a working point and it sets the tradeoff between electron and proton efficiencies, fake rates, and purities. A *receiver operating characteristic curve*, or ROC curve, is used to summarize the tradeoff between true positive rate (TPR) and false positive rate (FPR). Importantly, the characterization of the efficiency and rejection power (or equivalently the ROC curve) requires labeled data. In a particle physics context, it is recognized that the simulation is not perfect and the mismodeling is associated to the presence of systematic uncertainty. The discrepancy between the distribution of the training dataset and the distribution of the data where the model will be applied is referred to as *domain shift* or *distribution shift*. While mismodeling in the training dataset might lead to a suboptimal classifier, the real source of systematic uncertainty comes from the mismatch between the data used to characterize the performance of the classifier and the unlabeled data that the classifier is applied to. This motivates the use of data-driven methods to calibrate the resulting model.

[1] It is important to never use the testing dataset to make decisions about the model. For this purpose, another independent dataset, usually called the *validation dataset*, should be used (see Sec. 41.2.4).

This example provides a vertical slice through the various aspects of supervised machine learning in particle physics. Now we factorize and abstract the various ingredients in order to provide a more general treatment with a broader scope.

41.2 Supervised learning

Supervised learning generally refers to the class of problems where the training dataset are presented as input-output pairs $\{(x_i, y_i)\}_{i=1,\dots,n}$, where $x_i \in \mathcal{X}$ are the input features and $y_i \in \mathcal{Y}$ are the corresponding target labels. Furthermore, it is typically the case that x_i and y_i are independent and identically distributed (i.i.d.) according to the data distribution $p(x, y)$, $(x_i, y_i) \overset{\text{i.i.d.}}{\sim} p(x, y)$, though $p(x, y)$ is usually not known explicitly.

The goal of supervised learning is find a function $f : \mathcal{X} \to Y$ that 'best' captures the relationship between the input features and the corresponding target labels, similar to parameter estimation described in Sec. 40.2 of the Statistics chapter. Section 41.2.1 discusses how we quantify which function is best.

41.2.1 *Loss, risk, empirical risk*

The term *learning* in machine learning generally refers to optimization of some objective, which can be thought of as minimizing *risk*. The risk brings together three main ingredients. The first is the *model family* $\mathcal{F}$ (where $f \in \mathcal{F}$ is the quantity that we vary during optimization), the second is the *loss function* $\mathcal{L}$, and the third is a data distribution $p(x, y)$. The *risk* for a model $f \in \mathcal{F}$ is defined as its expected loss

$$\mathcal{R}[f] \coloneqq \mathbb{E}_{p(x,y)}[\mathcal{L}(y, f(x))] \equiv \int \mathcal{L}(y, f(x))\, p(x, y)\, \mathrm{d}x\, \mathrm{d}y \,, \tag{41.1}$$

where $\mathbb{E}_p[\cdot]$ refers to the expectation with respect to the distribution p. Written this way, the risk is a functional, and the idealized goal for machine learning is to solve the optimization problem

$$f^* = \arg\min_{f \in \mathcal{F}} \mathcal{R}[f] \,, \tag{41.2}$$

where $\mathcal{F}$ would include all possible functions.

One of the defining characteristics of machine learning in practice is that one does not know the data distribution $p(x, y)$, but does have access to samples from that distribution, *i.e.* $\{x_i, y_i\}_{i=1,\dots,n}$ with $(x_i, y_i) \overset{\text{i.i.d.}}{\sim} p(x, y)$. This leads to the corresponding *empirical risk*

$$\mathcal{R}_{\text{emp}}[f] \coloneqq \mathbb{E}_{\hat{p}(x,y)}[\mathcal{L}(y, f(x))] \equiv \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, f(x_i)) \,, \tag{41.3}$$

where $\hat{p}(x, y) = \frac{1}{n} \sum_{i=1}^{n} \delta(x - x_i)\delta(y - y_i)$ is referred to as the empirical distribution of the dataset $\{(x_i, y_i)\}_{i=1,\dots,n}$. The *empirical risk minimization* principle is a core idea in statistical learning theory [7], which approximates f^* with its empirical analogue

$$\hat{f} = \arg\min_{f \in \hat{\mathcal{F}}} \mathcal{R}_{\text{emp}}[f] \,, \tag{41.4}$$

where $\hat{\mathcal{F}}$ is the set of all possible functions parametrized by the model parameters ϕ. In an idealized infinite parameter limit machine learning functions, such as neural networks, are often universal approximators, meaning they cover all functions and $\hat{\mathcal{F}} = \mathcal{F}$. For finite size models, this may not be a valid approximation. Expressivity of the network characterizes this universality property and is a function of the network architecture and its parameters such as width and depth of neural network

layers. If the expressivity is too small it leads to underfitting. However, an equally important consideration is the risk of overfitting if we optimize Eq. 41.4 for too long or use an unrestricted model class (see Sec. 41.2.5).

While the loss function may quantify some well-motivated notion of risk, it is also common to design loss functions so that f^* has some desired property. In Secs. 41.2.2–41.2.5, we will consider several such loss functions where one can show that the corresponding f^* has the desired property even if the form of the loss is not obvious. Furthermore, there are often multiple loss functions that can lead to the same f^*. Thus, one can think of machine learning as solving Eq. 41.4 with a sufficiently flexible model, powerful optimization algorithms, and practical considerations to break the degeneracy between different loss functions that lead to the same f^*. As we shall see, commonly used loss functions can also be mathematically derived from a probabilistic approach.

41.2.2 *Regression*

The goal of regression is to predict a label $y \in \mathcal{Y}$ given an input feature vector $x \in \mathcal{X}$. Typically, the label is a real-valued scalar, but $\mathcal{X}$ can be $\mathbb{R}^d$ or some more structured target (*e.g.*, an image, sequence, graph, quantile, or distribution). When $\mathcal{Y}$ is discrete, the task is usually referred to as classification (see Sec. 41.2.3); however, the two are closely related and *logistic regression* is an example where the model predicts a continuous probability associated to the possible label values. In elementary statistical language, the target label y is often called a dependent variable, while the feature x is called the independent variable. In classical statistics, one often assumes a model for the data such as

$$y_i = f_\phi(x_i) + e_i \ , \tag{41.5}$$

where e_i is an additive error term that is often assumed to be independent of x and normally distributed. This leads to classic approaches like least-squares (see Sec. 40.2.3), and when the model f_ϕ is linear in ϕ (not in x!) linear regression, which has a closed-form solution. However, we can relax these assumptions and consider the general case of an arbitrary joint distribution $p(x, y)$, which can be written as $p(y|x)p(x)$ without loss of generality (see Sec. 39.1). Consider the *squared error* as a loss function, which corresponds to the mean-squared error (MSE) empirical risk:

$$\mathcal{L}_{\text{MSE}}(y, f(x)) = (y - f(x))^2 \ . \tag{41.6}$$

One might expect that the squared error would only be appropriate in the case that the conditional distribution $p(y|x)$ is normally distributed, but one can use the calculus of variations to show that in general

$$f^*_{\text{MSE}}(x) = \mathbb{E}_{p(y|x)}[y] \ , \tag{41.7}$$

that is the optimal regressor for the MSE is the conditional expectation of y given x.

One issue with the squared-error as a loss function is that it is very sensitive to outliers. Alternatively, one can use the absolute error $|y - f(x)|$ as a loss function[2]. However, the discontinuous derivative of the absolute (L1) error leads to challenges in optimization. As a result there are various other loss functions, such as the Huber loss, that aim to be both robust and more amenable to optimization.

Note that this framing of regression yields a function $f(x)$ that only provides a point estimate for y. An alternative approach to regression is to model the full conditional distribution $p(y|x)$. One such example is Gaussian process regression, which is discussed in Sec. 41.8.2. In that probabilistic approach, one can still obtain a point estimator, such as the conditional expectation or the maximum a posteriori (MAP) estimator

$$f^*(x) = \arg\max_y p(y|x) \ , \tag{41.8}$$

[2]The absolute error and squared error are often denoted as L1 and L2 errors, respectively, in reference to the corresponding norms.

and one can also derive uncertainty estimates on the predicted value y (see Sec. 41.10 for more details). In this setting, the prior distribution on the model family is closely related to the concept of regularization, which we touch on in Secs. 41.2.5 and 41.8.2.

When one directly models $p(y|x)$, or goes further to model the joint distribution $p(x, y) = p(y|x)p(x)$, then one can use maximum likelihood for the loss function. In that approach, the problem is really one of density estimation, which is a type of unsupervised learning that we discuss in Sec. 41.3.3. These two approaches are a classic examples of two different approaches to modeling. Regression with $f^*_{\text{MSE}}(x)$ is the prototypical example of *discriminative* modeling, while modeling the joint distribution is a prototypical example of *generative* modeling. Generally, discriminative approaches with supervised learning outperform generative approaches when there is sufficient data, but generative approaches can be beneficial in data-starved settings [8].

41.2.3 *Classification*

The goal of classification is to predict one of a finite number of class labels $y \in \mathcal{Y}$ given an input feature vector $x \in \mathbb{X}$. It is similar to regression in this way, but the focus is on discrete target space $\mathcal{Y}$. An important special case is when the label can only take on one of two values (*e.g.*, "signal" or "background"), which is referred to as binary classification and is equivalent to simple hypothesis testing in statistics. It is common for a classifier to be the composition of two functions: $f(g(x))$. The first function $g : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$ predicts continuous probabilities for each class (*i.e.*, $g_c(x) \approx p(y = c|x)$). The second function $f : \mathbb{R}^{|\mathcal{Y}|} \to \mathcal{Y}$ then chooses the discrete label $y \in \mathcal{Y}$, such as $f(g(x)) = \arg\max_c g_c(x) \approx \arg\max_y p(y|x)$. This is the case for both classical methods like logistic regression and modern, deep learning approaches to classification; therefore, we will use the term probabilistic classifier for $g(x)$ or just classifier when it is clear in context.

An intuitive loss function for classification is the zero-one loss, which simply counts the number of mis-classifications:

$$\mathcal{L}_{0/1}(y, f(x)) = \begin{cases} 0, & \text{if } f(x) = y \\ 1, & \text{otherwise .} \end{cases} \tag{41.9}$$

The zero-one loss can also be written as $\mathcal{L}_{0/1}(y, f(x)) = \mathbf{1}(y \neq f(x))$, where $\mathbf{1}(\cdot)$ is the indicator function. The zero-one loss is non-differentiable, so it does not pair well with gradient-based optimization.

For binary classification, one can use $y = \{0, 1\}$ as numerical values for the class labels and the *binary cross-entropy* loss function

$$\mathcal{L}_{\text{bxe}}(y, g(x)) = -\left[y \log(g(x)) + (1 - y) \log(1 - g(x))\right] . \tag{41.10}$$

The resulting model will approximate $f^*_{\text{bxe}}(x)$, which takes on the form

$$f^*_{\text{bxe}}(x) = \mathbb{E}_{p(y|x)}[y] = p(y = 1|x) = \frac{p(x|y = 1)p(y = 1)}{p(x|y = 0)p(y = 0) + p(x|y = 1)p(y = 1)} . \tag{41.11}$$

That is the binary cross-entropy loss for binary classification leads to the Bayesian posterior probability that the label $y = 1$ given the feature vector x (see Bayes theorem in Sec. 39.1).

Equation 41.11 highlights an important feature of supervised learning relevant for particle physics: the joint distribution $p(x, y)$ of the training dataset implies a prior distribution $p(y)$ on the labels or classes. This prior distribution reflects the frequency in the training dataset, not necessarily in the real data. When applying the resulting model to a different dataset with the same conditional distribution (data likelihood) $p(x|y)$ for the features and a different prior $p'(y)$ for the labels, the probabilistic interpretation of the result will not be properly calibrated, meaning $g(x) \not\approx p(y|x)$. A common choice for binary classification is to use a balanced training dataset with

$p(y=0) = p(y=1) = \frac{1}{2}$, while in many cases the true $p'(y=1)$ in the experimental data might be very small (*i.e.*, low signal-to-background), unknown, or zero (*i.e.*, a hypothetical particle that does not exist).

If $p'(y)$ and $p(y)$ are known then Bayes theorem can be used to re-calibrate the posterior $p(y|x)$ from one prior to another. One example of such re-calibration is the correspondence of binary classification to simple hypothesis tests in frequentist statistics discussed in Sec. 40.3.1 of the Statistics chapter. In that setting, the Neyman-Pearson lemma states that the optimal classifier is given by the likelihood ratio

$$f^*_{\text{NP}}(x) = \frac{p(x|y=1)}{p(x|y=0)} \,, \tag{41.12}$$

which does not depend on the prior probabilities $p'(y=0)$ or $p'(y=1)$ as in Eq. 41.11, or, equivalently, assumes equal priors $p'(y=0) = p'(y=1)$. Bayes theorem can be used to show that the two functions, $f^*_{\text{NP}}(x)$ and $f^*_{\text{bxe}}(x)$, are related by a one-to-one, monotonic transformation

$$f^*_{\text{NP}}(x) = \frac{p(y=0)}{p(y=1)} \frac{f^*_{\text{bxe}}(x)}{1 - f^*_{\text{bxe}}(x)} \,, \tag{41.13}$$

which is referred to as the *likelihood-ratio trick* and plays an important role in simulation-based inference (see Sec. 41.6).

A standard way to evaluate the performance of a classifier is to evaluate the true positive rate (TPR)—the proportion of $y=1$ samples that are correctly identified based on a fixed threshold $g(x) > c$—as a function of the false positive rate (FPR)—the proportion of $y=0$ samples that are misidentified based on the same fixed threshold. Plotting these values generates a graph known as the receiver operating characteristic (ROC) curve. Importantly, the monotonic transformation of Eq. 41.13 does not impact the tradeoff between FPR and TPR, therefore the ROC curves for $f^*_{\text{NP}}(x)$ and $f^*_{\text{bxe}}(x)$ are identical and do not depend on the prior probabilities $p(y)$. This property has been leveraged in *weakly supervised* approaches [9] to train a classifier in data without access to labels as long as one has two datasets with different $p(y=1)/p(y=0)$ ratios and the same conditional distribution $p(x|y)$ of the features given the labels.

A generalization of Eq. 41.10 that applies to multiple classes, is the *categorical cross-entropy* loss

$$\mathcal{L}_{\text{xe}}(y, f(x)) = -\sum_{c \in |\mathcal{Y}|} \mathbf{1}(y=c) \log(f_c(x)) \,, \tag{41.14}$$

where $f : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$ and the indicator function picks out the term in the sum for the corresponding class label y. This loss can be derived by maximizing the posterior of Eq. 41.26 using a discrete set of class labels y, which identifies $f_c(x) = \tilde{f}(y=c|x) = p(y=c|x)$ and thus assumes the constraint $\sum_c f_c(x) = 1$ and $f_c(x) \geq 0$ (see Sec. 41.8.4.2 for an activation function that enforces this). The function $\tilde{f}(y|x)$ can be interpreted as a conditional distribution, *i.e.*, an approximation to the true posterior $p(y|x)$. The risk associated to the cross entropy loss function is

$$\mathcal{R}_{\text{xe}}[f] = \mathbb{E}_{p(x,y)}\left[-\sum_{c \in |\mathcal{Y}|} \mathbf{1}(y=c) \log f_c(x)\right] = -\sum_{c \in |\mathcal{Y}|} p(y=c) \mathbb{E}_{p(x|y)}[\log \tilde{f}(y=c|x)] \,. \tag{41.15}$$

This is equivalent to $\mathcal{R}_{\text{xe}}[f] = \mathbb{E}_{p(x)}[H[p(y|x), \tilde{f}(y|x)]]$, where

$$H[p, f] \equiv \mathbb{E}_p[-\log f] = -\int p(x) \log(f(x)) \, \mathrm{d}x \tag{41.16}$$

is the cross entropy between the two distributions. One can use a Lagrange multiplier to enforce the normalization constraint and the calculus of variations to show that

$$f^*_{\text{xe},c}(x) = p(y = c|x) \ , \tag{41.17}$$

which is equivalent to the solution in Eq. 41.11 in the binary case.

This approach is closely related to the loss functions that are used for density estimation, the forward Kullback-Leibler (KL) divergence, and the maximum likelihood estimation. Minimizing cross entropy $H[p, f_\phi]$ to ϕ is equivalent to minimizing the forward KL divergence

$$\text{KL}(p\|f_\phi) \coloneqq \mathbb{E}_p[\log p(x)) - \log f_\phi] = H[p, f_\phi] - H[p] \ , \tag{41.18}$$

where $H[p] \coloneqq \int p(x) \log p(x) dx$ is the entropy and independent of f_ϕ. The KL divergence $\text{KL}[p\|f] \geq 0$, and equal if and only if $p = f$.

Unlike in the binary classification case, the multi-class classifier is sensitive to the priors $p(y)$ used in training. This leads to complications as often the class proportions are unknown. For example, one might be interested in classifying a signal when multiple backgrounds are present and the relative proportion of those different background components is uncertain. Ideally one would like the class proportions for the background components used in training to match those in the data, which presents an additional training challenge if those proportions are heavily unbalanced.

41.2.4 *Generalization and model complexity*

With a sufficiently flexible model, it is possible to fit the training dataset very well, though the model might not *generalize* well to unseen data, a phenomenon known as *overfitting*. More concretely, for a nonnegative loss function one might have $\mathcal{R}_{\text{emp}}[\hat{f}] \to 0$, while the true risk $\mathcal{R}[\hat{f}]$ might be large. Conversely, *underfitting* occurs when a model is unable to capture the relationship between the inputs and labels accurately, resulting in large empirical and true risks. While it is generally not possible to evaluate $\mathcal{R}[\hat{f}]$ exactly because we do not know $p(u)$, we can use an independent dataset (also called validation dataset) to obtain an unbiased estimate of it. This *cross-validation* method motivates the test-train-validation split of the data.

Intuitively, a model with many parameters has more flexibility and is more prone to overfitting. However, some highly over-parameterized models (that have large subspaces of their parameters where $\mathcal{R}_{\text{emp}}[\hat{f}_\phi] \to 0$) generalize well [10, 11]. Often this is achieved through *regularization*, both explicit and implicit (Sec. 41.2.5).

Two main sources of error prevent models from generalizing beyond their training dataset. One is *bias* arising from erroneous assumptions in the model and the other is *variance* arising from sensitivity to statistical fluctuations in the training dataset. The *bias-variance decomposition* is a way of analyzing a model's expected risk as a sum of bias and variance terms. Concretely, if $\mathcal{L}$ is the squared loss, one can decompose the expected risk $\mathbb{E}_\mathcal{D}[\mathcal{R}_{\text{emp}}[\hat{f}_\phi]]$ over all possible training datasets $\mathcal{D}$ into three terms [12, 13],

$$\mathbb{E}_\mathcal{D}\left[\mathcal{R}[\hat{f}^\mathcal{D}_\phi]\right] = \mathbb{E}_\mathcal{D}\mathbb{E}_{p(x,y)}\left[(y - \hat{f}^\mathcal{D}_\phi(x))^2\right] \tag{41.19}$$

$$= \mathbb{E}_{p(x)}\left[\underbrace{\mathbb{E}_{p(y|x)}\left[(y - \bar{y})^2\right]}_{\text{noise}} + \underbrace{\mathbb{E}_\mathcal{D}\left[(\hat{f}_\phi(x) - \bar{f}(x))^2\right]}_{\text{variance}} + \underbrace{(\bar{y} - \bar{f}(x))^2}_{\text{bias}}\right] \ , \tag{41.20}$$

where $\bar{f}(x) \equiv \mathbb{E}_\mathcal{D}\left[\hat{f}^\mathcal{D}_\phi(x)\right]$ is the "average" prediction of the model over different possible training datasets. In this expression, the first term is the inherent "noise" in the dataset, *i.e.*, the variance of y around its mean, which is zero if y is deterministically related to x. The second term is the

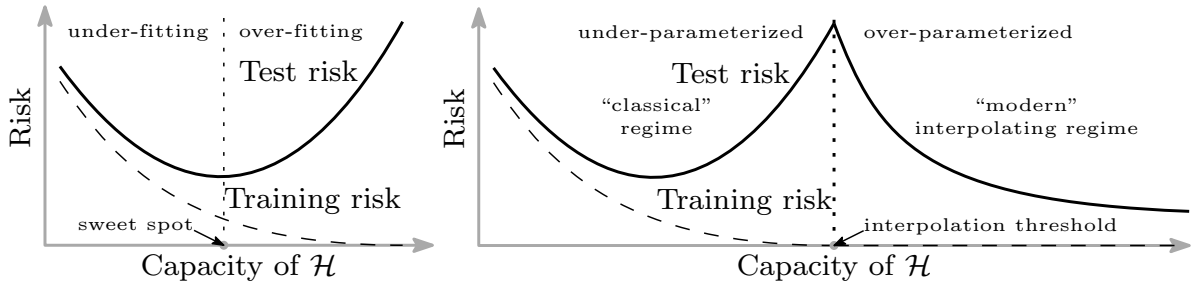

Figure 41.1: Curves for training risk (dashed line) and test risk (solid line) from Belkin et al. in Proceedings of the National Academy of Sciences, 2019. The classical U-shaped risk curve arising from the bias-variance trade-off (left) and the double descent risk curve (right), which incorporates the U-shaped risk curve (*i.e.*, the "classical" regime) together with the observed behavior from using high capacity function classes (*i.e.* the "modern" regime).

variance of the model around its average when considering different training datasets, and the third term is the squared bias, *i.e.*, the difference between the average prediction and the true conditional mean.

Classically, there is a correspondence between overfitting and underfitting and the concepts of bias and variance discussed in Sec. 40.2 on parameter estimation. 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. Thus, there exists a tradeoff between bias and variance, shown schematically in Fig. 41.1 (left).

However, in modern machine learning, very high-capacity models such as neural networks can be trained to exactly fit the data, and yet obtain high accuracy on test data [14], as shown in Fig. 41.1 (right). This phenomenon is known as "double descent." The apparent contradiction may be addressed by considering the regularizing (Sec. 41.2.5) effects of neural network training, specifically stochastic gradient descent (Sec. 41.9.2).

41.2.5 *Regularization*

The trained model $\hat{f}$, or equivalently, the parameters of the trained model $\hat{\phi}$ can be thought of as point estimates of f^*. The bias-variance tradeoff means that introducing a small bias can often lead to a significant reduction in variance. This motivates the explicit addition of a *regularization* term to the loss function, which will introduce some bias $f^*_{\text{reg}} \neq f^*$. A common form of regularization is to penalize by the L2 norm of the parameters (*i.e.* $\|\phi\|^2$), which is referred to as *L2 or Tikhonov regularization*. This appears in the form of penalized maximum likelihood, and it is also commonly used in unfolding [15]. Alternatively, one can penalize by the L1 norm $\|\phi\|$, which is known as *L1 regularization*. One can also interpret the regularization term as an explicit prior on the parameters, and the resulting model as the Bayesian maximum a posteriori (MAP) estimator. When L1 or L2 regularization is paired with linear regression, it is known as *LASSO regression* or *ridge regression*, respectively. In addition, L2 regularization paired with kernel machines gives rise to Gaussian process regression.

These two types of explicit regularization generally have solutions with different properties. For example, L1 regularization naturally induces sparsity, *i.e.*, a fraction of the parameters are nearly zero, whereas L2 regularization tends to keep all parameters nonzero but with lower magnitudes, as illustrated in Fig. 41.2. Because L1 regularization sets certain parameters to zero, it is often

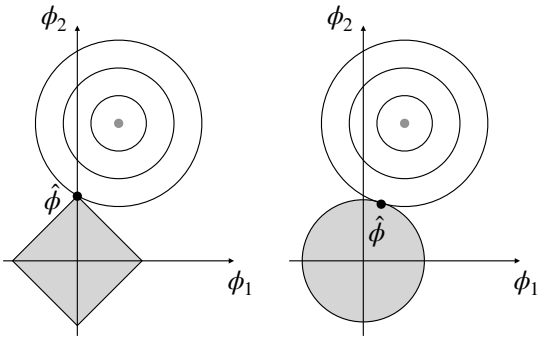

Figure 41.2: Depiction of L1 (left) and L2 (right) regularization constraint regions and the contours of an unregularized loss function. The intersection with the L1 constraint region gives an optimal value $\hat{\phi}$ that is sparse, *i.e.*, $\phi_1 = 0$, while the L2 contraint region yields an optimal value $\hat{\phi}$ where both ϕ_1 and ϕ_2 are small, but nonzero.

used as part of feature selection and model compression techniques, as discussed in Sec. 41.11.

Another form of regularization is to restrict the model class $\hat{\mathcal{F}}$. For example, a neural network and a sequence of narrow step functions (delta functions) can both be shown to be universal approximators in infinite parameter size limit, but on real world examples the former generalizes much better than the latter. Within the class of neural network models, convolutional neural networks are a subset of generic feedforward neural networks that approximately preserve translational symmetry (see Sec. 41.8.4.4 for more discussion). These types of choices are often encoded in the architecture of a neural network and are broadly referred to as *inductive bias* in the model.

In addition to explicit regularization terms in the loss function or through restrictions to the model class, it is also possible to regularize implicitly. One implicit regularization is through early stopping [15, 16], where we monitor the loss on the training dataset and the loss on held-out validation dataset. While the training loss continues to decrease with more gradient descent cycles, the validation loss may not, and early stopping stops the training when validation loss flattens out or begins to increase. Another powerful form of regularization used in deep learning models is known as *dropout* [17], which randomly removes some some parts of the model during training and can be thought of as implementing a type of model averaging [18].

The chosen numerical optimization procedure can also act as an implicit regularization. In the case of highly over-parameterized models where there is a large degenerate parameter space that achieves zero loss, $\Phi_0 = \{\phi | \mathcal{R}_{\text{emp}}[f_\phi]\} = 0$, the dynamics of the optimization algorithm will break the degeneracy and favor some particular $\hat{\phi} \in \Phi_0$ as if an additional regularization term was included. Despite zero loss and over-parametrization, the corresponding generalization error may be small, a phenomenon called *benign overfitting* [19]. Different optimization algorithms will have different implicit regularization effects, and thus favor different parameter points in Φ_0 that will have different generalization error [20]. Understanding this interaction is a topic of contemporary research in machine learning [21].

Some methods such as Gaussian process (GP) do not require optimization, and instead use linear algebra to obtain the solution. Benign overfitting is explicit for GP in that in the absence of noise the solution goes through all the training data, yet it generalizes well if the kernel is well chosen. Infinitely wide neural networks have an explicit correspondence to Gaussian process [22]. When applied to deep networks this leads to the concept of neural tangent kernel [23].

41.3 Unsupervised learning

Unsupervised learning generally refers to the class of problems that use unlabeled training dataset $\{x_i\}_{i=1,\dots,n}$, where $x_i \in \mathcal{X}$ are the input features. Furthermore, it is typically assumed that $(x_i) \overset{\text{i.i.d.}}{\sim} p(x)$, though $p(x)$ is usually not known explicitly. Finally, the loss function in unsupervised learning takes on the special form $\mathcal{L}(x, f(x))$. This class of learning has many different applications, such as density estimation, anomaly detection, generative learning, representation learning and clustering, each with the corresponding set of methods. Some of these tasks can be achieved with the same methods, *e.g.* normalizing flows (see Sec. 41.3.4.3) can perform density estimation, generative sampling and anomaly detection.

41.3.1 *Representation learning, compression, and autoencoders*

A recurring topic in machine learning and statistics is how to represent the data. Much of classical statistics involves constructing a low-dimensional summary statistic that extracts the relevant information from the data for a particular task (a sufficient statistic in the language of classical statistics). There is a spectrum of representations with tradeoffs. At one end of this spectrum is lossless compression that allows one to encode the data into a smaller, intermediate representation that carries all the information since it can be decoded back into the original data. At the other end of the spectrum is something like the likelihood ratio, which is a single scalar that carries the relevant information needed for hypothesis testing for a single hypothesis, but it discards all the other information that might be needed for other tasks, such as testing other hypotheses. An intermediate point in this spectrum is the process of feature engineering, which refers to the creation of new features $\mathcal{X}'$ from the original features $\mathcal{X}$ in hopes that the downstream task will be easier with the new features. For example, instead of working directly with the energy and momentum of particles, one might compute invariant masses or angles between particles. This type of feature engineering generally improves performance for shallow neural networks and decision trees; however, with the rise of deep learning this is often no longer necessary and may limit performance compared to working with the original features [4]. One can think of the intermediate layers of a neural network between the input and the output a representation of the data that is good for the task at hand, and by training all the layers of the network simultaneously (or "end-to-end") one can see the intermediate layers as a learned representations. For a review, see Ref. [24].

An example of a linear dimensionality reduction representation and data compression is principal component analysis (PCA) of data $x \in \mathbb{R}^d$ at fixed latent space dimensionality k ($k < d$), which finds the orthogonal linear transformation, O,

$$O : \mathbb{R}^k \to \mathbb{R}^d, z \mapsto Oz,\ OO^\intercal = I_d \tag{41.21}$$

that maximizes the data variance in the latent space. Maximizing the variance of the transformed data is equivalent to minimizing the average reconstruction error (the residual variance in data space),

$$\mathcal{L}_{\text{reco}}(x, f(x)) = \|x - f(x)\|^2 \,. \tag{41.22}$$

A PCA can thus be interpreted as a linear, orthogonal model that is trained to minimize the L_2-distance between the input data and the reconstructed data given the fixed dimensionality k. In practice, the PCA problem can be solved analytically without the use of optimization algorithms or the loss function: the principal components are given by the eigenvectors of the data covariance matrix.

A suitable latent space dimensionality, k, is chosen by ordering the eigenvalues, λ_i, of the data covariance in descending order, and keeping only the first few eigenvectors that correspond to the largest eigenvalues. The cut is often made at dimensionalities that capture around 90% of the data

variance. For many data sets this results in $k \ll d$. The average reconstruction error that originates from the discarded eigenvalues is $\sigma^2_{\text{reco}} = \sum_{i=k+1}^{d} \lambda_i$.

Another common type of representation learning and nonlinear dimensionality reduction is based on the *autoencoder* $f = g \circ e : \mathcal{X} \to \mathcal{X}$, where $e : \mathcal{X} \to \mathcal{Z}$ is referred to as the *encoder* and $g : \mathcal{Z} \to \mathcal{X}$ is referred to as the *generator* or *decoder*. Typically the dimensionality of $\mathcal{Z}$ is much less than $\mathcal{X}$, and $z = e(x)$ can be thought of as a compressed representation of the input. The intermediate space $\mathcal{Z}$ is sometimes referred to as the bottleneck or the latent space of the autoencoder. If the bottleneck is sufficiently large and the encoder and decoder are sufficiently flexible, then the function f could just be the identity (*i.e.*, lossless compression). However, if the encoder and decoder are not sufficiently flexible or the dimensionality of the latent space is not large enough there will be some reconstruction error. Therefore, the reconstruction error of Eq. 41.22 serves as a natural loss function of an autoencoder.

Once trained, the encoder $e(x)$ can be used independently of the decoder to provide a generic low-dimensional representation of the data. The flexibility of this approach is attractive; however, there are no guarantees that this representation will be optimal for the other task. Indeed, the transition from pre-trained autoencoders to end-to-end learning is one of the important trends that characterized the onset of the deep learning era.

While achieving zero reconstruction error may seem good as it would imply lossless compression, it often performs poorly in practice. First, the encoder may be overfit to the training dataset and not generalize well to held out data. This can be addressed by adding a prior to the training, discussed in Sec. 41.3.4.1. Second, it may not be robust to domain shift (see Sec. 41.10.2).

41.3.2 *Clustering*

The goal of clustering is to group the data $\{x_i\}_{i=1,\ldots,n}$ into k groups, or *clusters*, usually with $k \ll n$. Intuitively, if two data points belong to the same cluster, then they should be similar in some sense. Conversely, if two data points are very different, then they should be assigned to different clusters. The notion of similarity usually is based on some heuristic, and there are a variety of algorithmic and probabilistic clustering algorithms. In some cases k is specified, while in others it is determined by the clustering algorithm. There is also a distinction between flat clustering that directly partitions the data into k clusters and hierarchical clustering where clusters are nested hierarchically as the name suggests. In many cases, clustering uses some notion of distance $d(x_i, x_j)$, which may be the L_p norm $\|x_i - x_j\|_p$.

One of the most common clustering algorithms is known as k-means, where k is specified by the user and results in sets $S = \{S_1, \ldots, S_k\}$ that minimize the variance of each cluster. Thus, the objective is

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{\mathbf{x},\mathbf{y} \in S_i} \|\mathbf{x} - \mathbf{y}\|^2 \tag{41.23}$$

where μ_i is the mean of points in S_i. k-means can be interpreted as a Gaussian mixture density estimation of $p(x)$, where all the Gaussians are isotropic. It can be generalized to a Gaussian mixture model, where both the means and the covariance matrix are estimated.

Among the other class of algorithms that determine k, density-based spatial clustering of applications with noise (DBSCAN) is one of the most frequently used. DBSCAN clusters points based on a distance metric (*e.g.*, Euclidean) defined for each application. Two hyperparameters are ϵ, the maximum distance threshold to determine whether a neighboring point belongs to the same cluster, and the minimum sample size for a group of close points to be identified as a valid cluster or noise. While DBSCAN is robust against irregularly shaped clusters with a simple distance-based metric, single threshold parameter ϵ shared to distinguish all clusters can be challenging. Hierarchical

DBSCAN (HDBSCAN) generalizes to varying densities by building a hierarchy of density-based clusters across all ϵ via mutual-reachability distances, then extracts the most stable clusters from a condensed tree.

Finally, neural networks are often used for clustering in particle physics. One use case is to transform the data points into a latent space where clustering is performed using an unsupervised, traditional algorithm. For example, an input dataset may not follow an isotropic gaussian distribution which is assumed by k-means, but one can design a neural network to learn a transformation into the latent space where this assumption holds. Another use case is to use neural network directly for clustering operation. Examples include object detection [25] and segmentation [26, 27] in computer vision (see Sec. 41.8.4.4) as well as clustering of graph nodes via edge classification [28–32] (see Sec. 41.8.4.7).

41.3.3 *Density estimation*

The goal of density estimation is to estimate a distribution $p(x)$ based on samples $\{x_i\}_{i=1,\dots,n}$ with $x_i \overset{\text{i.i.d.}}{\sim} p(x)$. Conceptually, this is the same goal as when fitting a parameterized distribution $f(x;\theta)$ to data using the method of maximum likelihood as described in Sec. 40.2.2 of the chapter on statistics. In practice, the difference in the machine learning context has to do with the flexibility of the model and the dimensionality of the data. A highly-flexible model, which can effectively approximate any distribution, is referred to as a non-parametric model (though, ironically, usually this means the model has many parameters). In contrast, typical maximum likelihood fits in particle physics are based on restricted families of distributions with relatively few parameters and the data is typically one- or two-dimensional, though occasionally five- or six-dimensional.

Maximizing the likelihood function in Eq. 40.10, $\mathcal{L}(\theta) = \prod_{i=1}^{n} f(x_i;\theta)$ is equivalent to minimizing the empirical risk:

$$\mathcal{R}_{\text{emp,xe}}[f_\phi] = -\frac{1}{n}\sum_{i=1}^{n} \log f_\phi(x) \,, \tag{41.24}$$

where we adopt the notation used in this chapter. The loss is simply $\mathcal{L}(x, f_\phi(x)) = -\log f_\phi(x)$, and the corresponding risk is

$$\mathcal{R}_{\text{xe}}[f_\phi] = \mathbb{E}_{p(x)}[-\log f_\phi(x)] \,, \tag{41.25}$$

which is the cross entropy $H[p, f_\phi]$. For density estimation, the model is usually constructed to enforce $\int f_\phi(x)dx = 1$ and $f_\phi(x) \geq 0$ so that it can be interpreted as a distribution. With this constraint, one can show that $f^*_{\text{xe}}(x) = p(x)$. This is not the only form of training: flow matching and diffusion methods train on a different objective, discussed further below.

The concepts of generalization and overfitting are particularly acute in *unsupervised* learning, where the likelihood maximization of equation 41.24, combined with universal approximator assumption, must converge onto $\hat{p}(x) = \frac{1}{n}\sum_{i=1}^{n} \delta(x - x_i)$, the empirical distribution of the dataset $\{x_i\}_{i=1,\dots,n}$. This distribution has the highest likelihood on the training dataset and the lowest likelihood on the test data where it gives $\hat{p}(x) = 0$ as long as the test dataset are not identical to the training dataset. So the empirical distribution of the training dataset has the worst possible generalization property, yet it is the solution we converge to for sufficiently expressive architectures in the absence of any regularization. In contrast, in supervised learning we often observe the phenomenon of benign overfitting, where even zero loss can generalize well.

In addition to approaches to density estimation that involve learning in the sense of minimizing a loss or risk function, we note that there are also classical density estimation techniques such as histogramming and kernel density estimation [33–35]. These techniques often fail in very high dimensions.

41.3.4 *Generative models*

Deep generative models are powerful machine learning models that can learn complex, high-dimensional distributions and generate samples from them. Because of their inherently probabilistic formulation, generative models are rapidly becoming an indispensable tool for scientific data analysis in a range of domains. The goal of generative models is to draw samples from $p(x)$. For some formulations of learned $p(x)$, such as normalizing flows, the samples can be drawn directly. For other explicit formulations of $p(x)$, such as Boltzmann machines, one can use sampling techniques such as Monte Carlo Markov chain sampling. There are however many other approaches to drawing samples from $p(x)$ that do not rely on its explicit form.

Generative models can be contrasted against discriminative models that are primarily used for supervised learning tasks. Roughly, discriminative models are used for prediction and $f(x)$ provides a point estimate of the target y, and they are more closely connected to function approximation. In contrast, generative models describe the data distribution $p(x)$ (or the joint data distribution $p(x, y)$ in a supervised setting). An enlightening discussion of these two approaches can be found in Ref. [8].

There are a number of different types of deep generative models that have various pros and cons as they do not all have the same capabilities. We will focus on variational autoencoders (VAEs) [36, 37] , generative adversarial networks (GANs) [38, 39], normalizing flows (NFs) [40–44], and flow-matching and diffusion models [45–48], though other approaches have been explored in this quickly developing area of research. Consider these three distinct types of functionality:

- **generation:** ability to sample or "generate" a data point $x_i \sim p(x)$.
- **likelihood for generated data:** ability to evaluate the probability density (likelihood) $p(x_i)$ for a data point x_i sampled from the model $x_i \sim p(x)$.
- **likelihood for arbitrary data:** ability to evaluate the probability density $p(x_i)$ for an arbitrary data point $x_i \in \mathcal{X}$.

Each of the models above can be used for generation; however, only normalizing flows provide all three capabilities. For reasons that we will describe below, GANs and VAEs do not provide a tractable likelihood function, and they are sometimes referred to as *implicit models*. This establishes a connection to simulation-based inference where most scientific simulators are also implicit models with an intractable likelihood. Because normalizing flows have a tractable likelihood, they can be trained via maximum likelihood (Eq. 41.24) as described in Sec. 41.3.3. GANs and VAEs, on the other hand, need to employ some other loss function to be trained. In the case of VAEs, training is based on the ELBO used in variational inference (see Sec. 41.2.3 and the discussion around the reverse KL divergence below Eq. 41.18). While GANs are also implicit models they data they can generate is typically restricted to a lower-dimensional manifold $\mathcal{M} \subset \mathcal{X}$, meaning that almost all real training dataset doesn't "live on" the subspace of possibilities that the model can produce. In this case, the likelihood is for almost all data is zero, and so even ELBO-based training will not work. The breakthrough idea introduced in Ref. [38] was to use adversarial training where a classifier would be used to quantify how different the data generated from the model is from the data from the target distribution.

VAEs, GANs, and normalizing flows introduce a mapping $g(z, \theta)$ from a base random variable z to the space of the data $\mathcal{X}$. The map $g(z, \theta)$ is typically implemented with a neural network. The random variable z is sampled from some known base distribution $p(z)$ that is both easy to sample and has a density that is easy to evaluate. Typically, the base distribution is a multivariate normal.

In the literature on GANs and normalizing flows, this base random variable is often referred to as a latent variable and $p(z)$ is often referred to as a prior distribution. In the case of VAEs, one additionally adds some normally-distributed (Gaussian) random noise ϵ to the output so that

$x = g(z, \theta) + \epsilon$. In this case, x and z are not deterministically related and z is a legitimate latent variable in the model and $p(z)$ can be interpreted as the prior on that latent variable. In this case, the model can populate the full space of the data. Unfortunately, the marginal likelihood $p(x) = \int p(x, z)dz$ involves an intractable integral, thus maximum likelihood training is infeasible. However, the likelihood term $p(x|z)$ is tractable (*i.e.* the Gaussian noise), so training with the ELBO is possible.

Note that the dimensionality of z need not be the same as that of x. If $z \in \mathbb{R}^q$ and $\mathcal{X} = \mathbb{R}^d$ with $q < d$, then all points $g(z, \theta)$ will lie on a d-dimensional surface in $\mathbb{R}^d$. In the case of a VAE, the Gaussian noise ϵ means that the generated data x will be distributed in a thin region around the surface defined by $g(z, \theta)$. The presence of a bottleneck (*i.e.* $q < d$) leads to advantages and disadvantages. The disadvantages for GANs is that the likelihood assigned to almost all real world data (*i.e.* data not generated by the model) will be zero, so training is more difficult and many tasks in probabilistic inference won't be applicable. However, often real world data is also effectively described by a low-dimensional subspace in the full space of the data – random images look like noise, while natural images are in some sense special. For this reason, images produced by GANs for instance often have better visual quality than those produced by other techniques. This points to the ambiguity encountered in quantifying how close two distributions are, and also motivates the use of distance measures such as the Earth movers distance or Wasserstein distance [49, 50]. Conversely, the lack of a bottleneck (*i.e.* $q = d$) leads to very large models and scalability issues when the data is high dimensional.

Recent work has also focused on combining ideas from VAEs, GANs, and normalizing flows so that the generative model does involve a bottleneck but can still provide tractable likelihoods for density estimation restricted to that manifold [40, 51–54]. Some of these models can also be used in the context of anomaly detection and out of distribution detection by identifying data that is off the manifold.

The parametrization of the mapping (the architecture of the neural network) should match the structure of the data and be expressive enough. For problems with explicit symmetries it is beneficial to include them into the architecture of the network explicitly, which restricts the allowed space of the models and matches their inductive bias (implicit regularization inherently built into the choice of architecture of the network) to the data. Different architectures have been proposed [43, 55–57], and to achieve the best performance on a new dataset one needs extensive hyperparameter explorations [58].

41.3.4.1 *Variational autoencoders*

The autoencoder was described in Sec. 41.3.1 as model for compression and representation learning. The model is $f = g \circ e : \mathcal{X} \to \mathcal{X}$, where $e : \mathcal{X} \to \mathcal{Z}$ is referred to as the *encoder* and $g : \mathcal{Z} \to \mathcal{X}$ is referred to as the *generator* or *decoder*. The standard autoencoder is not a probabilistic model, but additional probabilitic structure can be added.

One approach is VAE mentioned above [36, 37]. By equipping the latent space with a prior distribution $p(z)$, the decoder of the autoencoder $g(z, \theta)$ implies a distribution on a manifold in the output space $\mathcal{X}$. VAEs additionally add some normally-distributed (Gaussian) random noise ϵ to the output so that $x = g(z, \theta) + \epsilon$. This implies that $p_\theta(x|z)$ is a tractable quantity, and it is interpreted as the likelihood in this context.

In a VAE one also elevates the encoder to have a probabilistic form. Instead of encoding $z = e(x)$ in a deterministic way, one seeks a distribution over z given x. A natural target for the probabilistic encoder would be to probabilistically invert the decoder. This inverse problem is

solved by the posterior distribution $p(z|x)$ via Bayes theorem

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \,. \tag{41.26}$$

While the likelihood and the prior may both be tractable, the normalizing constant $p(x) = \int p(x,z)dz$ involves an intractable integral (the same intractable integral that makes maximum likelihood training of the VAE infeasible).

One approach to Bayesian inference in these settings is variational inference (VI). In VI one approximates the posterior with some parametric family $q_\phi(z|x)$ in a parametric form, and then optimizes the ELBO with respect to its parameters ϕ.

$$\text{ELBO} = \mathbb{E}_{q(z)} \log p(x|z) - D_{\text{KL}}[q(z)||p(z)] \leq \log \mathbb{E}_{q(z)} \left[\frac{p(x,z)}{q(z)} \right] = \log p(x) \,, \tag{41.27}$$

where we used Jensen's inequality for concave functions (log) and the reverse Kullback-Leibler (KL) divergence term is

$$D_{\text{KL}}[q(z)||p(z)] = \mathbb{E}_{q(z)}[\log q(z) - \log p(z)] \geq 0 \,. \tag{41.28}$$

In a VAE, the variational model for the posterior $q_\phi(z|x)$ is often assumed to be an uncorrelated Gaussian (this is often called mean field approximation) defined by the mean μ and variance Σ. Instead of optimizing the mean and variance independently for each x, VAEs use neural networks to predict the mean $\mu_\phi(x)$ and the variance $\Sigma_\phi(x)$. This is called *amortized inference*, since after an up-front training cost the approximate posterior $q_\phi(z|x)$ can be evaluated efficiently with a single forward pass of the neural network. Note the standard auto-encoder is recovered if one only used the mean $\mu_\phi(x)$ for the encoder and did not add noise ϵ to the decoder.

Both the probabilistic encoder $q_\phi(z|x)$ and the probabilistic decoder $p_\theta(x|z)$ are trained jointly by optimizing the ELBO. Unlike the standard autoencoder, which only minimizes the reconstruction error, ELBO optimization of Eq. 41.27 has a tradeoff between minimizing the reconstruction error in the first term (averaged over the approximate posterior $q(z)$), which encourages high quality reconstructions, and minimizing the KL divergence term, which forces the posterior $q(z)$ to be as close to the chosen prior $p(z)$, and thus controls the sample quality by matching the aggregate posterior with a chosen prior distribution [59]. This term regularizes the VAE latent space, such that every sample drawn from the prior $p(z)$ correspond to a valid sample. Successful VAE training requires to find a delicate balance between the two contributing terms to the ELBO. Whether the VAE training process succeeds in striking this balance depends on a number of factors, including the network architectures, the chosen prior and the class of allowed posterior distributions. Once trained, the VAE can be used as a generative model by sampling from the prior $z_i \sim p(z)$ and then decoding according to $p_\theta(x|z) = g(z,\theta) + \epsilon$.

VAEs allow for expressive architectures, enjoy the benefits of regularization through data compression and have a firm theoretical foundation. Compared to GANs [38] 41.3.4.2, VAEs are of particular interest to the scientific community as they provide a lower bound to the marginal likelihood (albeit potentially with a large gap) and a posterior distribution for the latent variables.

It is also interesting to consider a special case of the autoencoder and VAE where the encoder and decoder are restricted to be linear transformations, which is effectively PCA. In PCA the (linear) decoder can be written $g(z) = Oz$, where O is a matrix. As in the case of the autoencoder, PCA is not a probabilistic model, but probabilistic structure can be added. Probabilistic PCA [60] assumes that the latent variables follow a Gaussian distribution with mean zero and covariance Λ, where Λ is a diagonal matrix with the rank-ordered eigenvalues λ_i along its diagonal. The true distribution of the PCA components may be non-Gaussian, but a Gaussian is the maximum entropy

approximation given their first two moments. Note that in probabilistic PCA these moments are measured on training dataset (when finding the principal components).

One can generalize probabilistic PCA to use nonlinear encoder and decoder as in an autoencoder. A Gaussian prior is a poor ansatz for the latent space distribution of data proceed by an autoencoder. Instead one can learn the density of the training samples in latent space using a normalizing flow. This model was introduced in $\mathcal{M}$-flows [53] and in probabilistic autoencoder (PAE) [54], which achieves similar performance to a VAE in terms of sample quality without explicit ELBO optimization. In all these cases the dimensionality of the latent space is a hyperparameter to be chosen or optimized by the user. Unlike a standard VAE, these models do not add noise to the decoded output, thus the data is strictly restricted to the manifold defined by the decoder $g(z, \theta)$. However, unlike a GAN there is a well defined way to take an arbitrary data point x, project it onto the manifold, and calculate the density of the data point projected onto the manifold. Thus these models can also be used in the context of anomaly detection and out of distribution detection by identifying data that is off the manifold.

41.3.4.2 *Generative adversarial networks*

GANs [38] also typically choose a low dimensional latent space z with a known prior distribution $p(z)$, typically a normal (Gaussian) distribution with zero mean and unit variance. GANs do not add noise to the output $g(z, \theta)$, so the likelihood $p(x|z)$ (and marginal likelihood $p(x)$) for almost all of the data space is 0, which precludes training by maximum likelihood and the ELBO. Instead of training on ELBO, GANs train on a dissimilarity measure defined implicitly by a discriminator $D(x)$ (also referred to as the critic). Calculating the dissimilarity often involves it's own learning problem (*i.e.*, adversarial training of the discriminator).

The training is usually framed as a mini-max game

$$\min_g \max_D \mathcal{L}_{\text{GAN}} = \min_g \max_D \{\mathbb{E}_{x\sim p(x)} \log D(x) + \mathbb{E}_{z\sim p(z)} \log[1 - D(g(z))]\}. \tag{41.29}$$

The goal of the discriminator is to distinguish between true and generated data, hence we want to maximize this loss with respect to D, assigning 1 to true data and 0 to generated data. The goal of generator is to fool the discriminator such that it cannot distinguish between true and generated data, hence we want to minimize this loss with respect to g at fixed D. This can be viewed as a game theoretical setup in a zero sum game between generator and discriminator.

Instead of this game theory interpretation we can view the internal objective $\max_D \mathcal{L}_{\text{GAN}}$ as an implicit loss function that measures the dissimilarity between the target and generated distributions. The loss of Eq. 41.29 corresponds to the Jensen-Shannon (JS) divergence, which is a symmetrized form of KL divergence. However, JS divergence is hard to directly work with, and the adversarial training could bring many problems such as vanishing gradient, mode collapse (tendency of generator to cluster the samples around the training samples, with holes between them) and non-convergence [49, 50]. One of the core issues is that the distribution generated by the GAN is not guaranteed to cover the entire space. To address these issues Wasserstein GANs train on

$$\min_g \max_D \mathcal{L}_{\text{WGAN}} = \min_g \max_D \{\mathbb{E}_{x\sim p(x)} D(x) - \mathbb{E}_{z\sim p(z)} D(g(z))\}. \tag{41.30}$$

Here again the goal of discriminator is to make the loss as large as possible between the true data and the generated data, while the goal of generator is to make it as small as possible, so that the discriminator cannot distinguish between the two. There is no requirement for $D(x)$ to be between 0 and 1, which helps with the above mentioned problems of JS divergence. Instead, this is replaced with a requirement that $D(x)$ is 1-Lipshitz, i.e. the absolute value of the norm of the gradient of the discriminator output with respect to the input has to be less or equal to 1.

Eq. 41.30 can be interpreted as the dual form of the 1-Wasserstein distance between the true and generated distribution [61]. Wasserstein distances are a measure of dissimilarity between two distributions used in the context of optimal transport, a mathematical theory of how to define a notion of distance between probability distributions. Since the transport distance increases with the separation between the two distributions when they are non-overlapping, there is no gradient collapse that plagues other measures. In its primal form p-Wasserstein distance, $p \in [1, \infty)$, between two probability distributions p_1 and p_2, is defined as $W_p(p_1, p_2) = \inf_{\gamma \in \Pi(p_1,p_2)} \left(\mathbb{E}_{(x,y)\sim\gamma}\left[|x-y|^p\right]\right)^{\frac{1}{p}}$, where $\Pi(p_1, p_2)$ is the set of all possible joint distributions $\gamma(x, y)$ with marginalized distributions p_1 and p_2. In 1D the Wasserstein distance has a closed form solution via cumulative distribution functions (CDFs), but this evaluation is intractable in high dimensions.

In the dual form of 1-Wasserstein distance, one instead maximizes Eq. 41.30 over all possible functions $D(x)$ that are 1-Lipschitz. One way to implement this is through weight clipping of the parameters of discriminator network, but a simpler solution is to add a gradient norm penalty term explicitly to the loss function [62].

Because of the discriminative nature of the dissimilarity measure defined in data space, GANs and Wasserstein GANs often generate more realistic samples than VAE or normalizing flows in high dimensions such as natural images (although flow matching and diffusion models can outperform GANs). However, GANs do not provide an encoder from data to latent space nor a tractable likelihood $p(x)$.

41.3.4.3 *Normalizing flows and autoregressive models*

Normalizing flows (NFs) provide a powerful framework for density estimation and sampling [40–44, 63]. These models map the data x to latent variables z through a sequence of invertible transformations $f = f_1 \circ f_2 \circ \cdots \circ f_n$, such that $z = f(x)$ or $x = g(z) = f^{-1}(x)$. As in the VAE and GAN, z is modeled as a random number with a simple base distribution $p_Z(z)$, which is typically chosen to be a standard normal (Gaussian) distribution. Since NFs are invertible the dimensionality of the latent space equals the dimensionality of the data space, in contrast to VAE and GANs where the latent space dimensionality is often lower. The probability density of the model be evaluated using the change of variables formula:

$$p_X(x) = p_Z(f(x)) \left|\det\left(\frac{\partial f(x)}{\partial x}\right)\right| = p_Z(f(x)) \prod_{l=1}^{n} \left|\det\left(\frac{\partial f_l(x)}{\partial x}\right)\right|, \tag{41.31}$$

where we have added subscripts to $p_X(x)$ and $p_Z(z)$ for clarity. The Jacobian determinant $\det(\frac{\partial f_l(x)}{\partial x})$ must be efficient to compute for density estimation to be practical, and the transformation f_l should be easy to invert for sampling. In contrast to VAE and GANs, standard normalizing flows preserve the dimensionality of the data space as they are invertible (though there are normalizing flows that are defined on lower dimensional manifolds embedded in the data space [40, 51–54]). As such, unlike GANs and VAEs, they can be trained via maximum likelihood (Eq. 41.24) as described in Sec. 41.3.3.

There are several popular architectures of NFs. A method used by NICE, RealNVP and Glow [41–43] is to split the space into two disjoint sets z_1 and z_2, and then use an identity forward map $z \to x$ for x_1, $x_1 = z_1$, and an affine transformation for x_2 of the form

$$x_2 = \exp(s(z_1)) \odot z_2 + m(z_1), \tag{41.32}$$

where $\odot$ is elementwise product and $m(z_1)$, $s(z_1)$ are neural networks. The Jacobian of this map is lower triangular, and its determinant is simply the product of elements along the diagonal, which is tractable, as is the inverse of the transformation. At the next layer one then performs a

different split of dimensions into z_1 and z_2. The affine transformation can be further generalized to a nonlinear form using rational splines [64].

One can interpret the sequence of invertible transformations $f_1 \circ f_2 \circ \cdots \circ f_n$ as n discrete time steps in a continuous flow. In particular, one can think of a continuous-time flow described by an ordinary differential equation (ODE) and then interpret the discrete time steps as the result of a numerical integration of that ODE. This is the approach taken by the Ffjord algorithm [65] and other variants. A residual flow has an update $f_i(x) = x_i + \delta u_i(x)$, which for $\delta_i = n^{-1}$ and taking $n \to \infty$ limit gives rise to an ordinary differential equation (ODE)

$$dx_t = u_t(x_t)dt. \tag{41.33}$$

Here u_t is the velocity field that defines the flow and is a vector field. One can build the density estimator for all intermediate times t $p_t(x)$ using its divergence,

$$\ln p_t(x_t) = \ln p_0(x_0) - \int_0^t \nabla \cdot u_s(x_s)ds, \tag{41.34}$$

where p_0 at t_0 is the initial base distribution and p_1 at $t = 1$ is the target distribution. Continuous normalizing flows parametrize u_t as a neural network. They are very expressive, but expensive to train using maximum likelihood.

A different approach creating a deep generative model with a tractable likelihood is to model $p(x)$ autoregressively as

$$p(x) = \prod_{i=1}^{n} p(x_i|x_1, x_2, \ldots, x_{i-1}) \ . \tag{41.35}$$

This form describes each new dimension conditionally on all previous dimensions. It can model a general likelihood $p(x)$ as a sequence of conditional 1d distributions, whose conditional dependence on the parameters $x_1, x_2, \ldots, x_{i-1}$ can be modeled with neural networks. If x is a time series this form imposes a causal structure where x_i depends on all previous times x_j, $j < i$. WaveNet [66] and PixelCNN [67]) are two well known examples. Sampling from an autoregressive model is sequential, and can be slow in high dimensions. Inverse autoregressive flow reverses this process and makes sampling fast, but the likelihood evaluation is slow. Some normalizing flows have autoregressive coupling layers, such as masked autoregressive flow (MAF) [63].

All of the methods above use maximum likelihood training of likelihood $p(x)$ against network parameters, so the training is to minimize KL divergence between the data distribution and a Gaussian in latent space. This can be overly sensitive to small variance directions that dominate the likelihood, without being sensitive to the global structure of the data. An alternative is to use Optimal Transport Wasserstein distance between the density of the generated samples and the data, which can be evaluated either in data space or in latent space. As Wasserstein distance is difficult to evaluate in high dimensions, one can instead use slices, 1d projections of the data along different directions in high dimensional space, to build the flow [68]. Because this training is less sensitive to small variance directions than maximum likelihood training it achieves better results on anomaly detection tasks [68].

We end by noting that normalizing flows, autoregressive models, and other deep generative models that provide a tractable likelihood are powerful tools for simulation-based inference. They can provide surrogate models trained from large simulated datasets when the simulators have intractable likelihood functions, which is usually the case. As described in Sec. 41.6, one would like to work with models that can provide conditional density estimation in order to model either the likelihood $p(x|\theta)$ or the posterior $p(\theta|x)$ [69, 70]. These techniques are being actively explored and applied to a number of scientific problems.

41.3.4.4 *Flow-matching and diffusion models*

In flow-matching models, we start from a base distribution such as a Gaussian $p_0 = \mathcal{N}(0, I)$, and use ODEs to generate samples with a flow vector field $u_t^\theta(x)$ as in Eq. 41.33. As discussed above, continuous normalizing flows are expensive to train via maximum likelihood. Instead, one can learn directly the velocity field parametrized as a neural network $u_t^\theta(x_t)$ with parameters θ via the flow-matching loss

$$\mathcal{L} = \mathbb{E}_{t\sim U(0,1),\ x\sim p_t} \left[u_t(x_t) - u_t^\theta(x_t)\right]^2, \tag{41.36}$$

where the expectation is uniform over time t and over all intermediate distributions p_t. This equation is however not practical since we do not know the target $u_t(x)$. Instead, one can take advantage of the target conditional velocity field $u_t(x_t|z)$, where $z \sim \hat{p}$ is a training data sample

$$\mathcal{L} = \mathbb{E}_{t\sim U(0,1),\ x\sim p_t,\ z\sim\hat{p}} \left[u_t(x|z) - u_t^\theta(x_t)\right]^2. \tag{41.37}$$

It has been shown that this conditional target velocity field training also leads to the correct distribution in the flow models [45, 48]. Figure 41.3, taken from Ref. [71], illustrates the main idea, which is that training a conditional flow, and averaging over all the training data, is the same as training on unconditional flow.

The advantage of this formulation is that conditional velocity fields are a lot simpler to construct. A typical case is a flow from the initial Gaussian $p_0(x|z) = \mathcal{N}(0, I)$ to a delta function at z $p_1(x|z) = \delta_z(x)$. A very simple linear flow that achieves this is $p_t(x|z) = \mathcal{N}(tz, (1-t)^2 I)$. The flow itself moves from a random Gaussian variable $\epsilon \sim \mathcal{N}(0, I)$ to the data point z, so $x_t = \epsilon(1-t) + tz$. Finally, the conditional velocity field is given by $u(x_t|z) = z - \epsilon$, so the training loss is

$$\mathcal{L} = \mathbb{E}_{t\sim U(0,1),\ \epsilon\sim\mathcal{N}(0,I),\ z\sim\hat{p}} \left[z - \epsilon - u_t^\theta(\epsilon(1-t) + tz)\right]^2. \tag{41.38}$$

This leads to a simple training algorithm where one randomly chooses a minibatch of data z, random Gaussian variables ϵ, and a time t to update the parameters θ based on stochastic gradient descent using the loss of Eq. 41.38. The simplicity and efficiency of this training procedure has led flow matching to become one of the leading generative models for large image based data. Sampling from flow-matching models requires randomly choosing an initial condition $x_0 \sim N(0, I)$ and discretizing Eq. 41.33. Note that this is a deterministic ODE and all the randomness is in the initial conditions.

Diffusion models also start from a Gaussian base distribution, but also continuously add noise during the evolution in time, *i.e.*, they are based on a stochastic differential equation (SDE)

$$x_0 \sim p_0, \ \ dx_t = u_t^\theta(x_t)dt + \frac{\sigma_t^2}{2} s_t(x_t) + \sigma_t dW_t, \tag{41.39}$$

where we define score $s_t(x_t) = \nabla \ln p_t(x_t)$. Here, dW_t is the stochastic term, which adds Brownian motion (also called a Wiener process) in the form of uncorrelated Gaussian random noise. Note that with $\sigma_t = 0$, a diffusion model becomes a flow model. The noise variance σ_t is a free parameter that can be tuned for optimal performance. If our target is a static distribution so that $p_t = p$ then $u_t = 0$ and we obtain the Langevin equation.

In diffusion, we also need to learn the gradient field via the score function, and as before we can replace the marginal score with conditional score during the training to obtain score matching training procedure

$$\mathcal{L} = \mathbb{E}_{t\sim U(0,1),\ x\sim p_t,\ z\sim\hat{p}} \left[\nabla \ln p_t(x|z) - s_t^\theta(x_t)\right]^2. \tag{41.40}$$

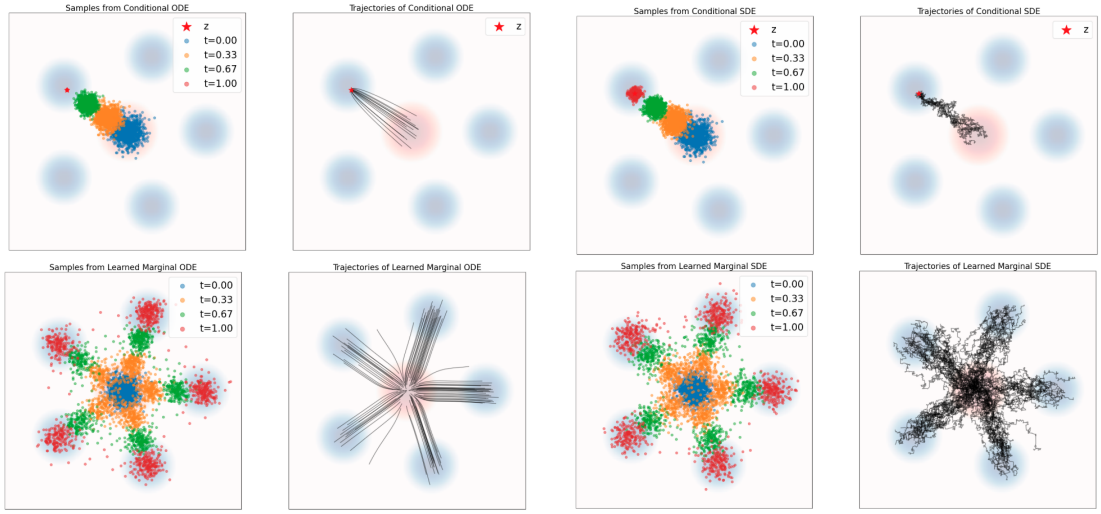

Figure 41.3: Illustration of the marginalization trick for flow-based models (left) and diffusion models (right), which simulate a probability path with ODEs or SDEs, respectively (Holderrieth and Erives, 2025). The data distribution $\hat{p}$ is the blue background, while the initial Gaussian distribution is the red background. The top graphs represent conditional probability paths, while the bottom graphs represent marginal probability paths. Both samples and trajectories are shown.

In the simple Gaussian example with $p_t(x_t) = \mathcal{N}(\alpha_t z, \beta_t^2 I)$, where $\alpha_0 = \beta_1 = 0$ and $\alpha_1 = \beta_0 = 1$, we have a trajectory $x_t = \alpha_t z + \beta_t \epsilon$, and the score loss

$$\mathcal{L} = \mathbb{E}_{t\sim U(0,1),\ \epsilon\sim\mathcal{N}(0,I),\ z\sim\hat{p}} \left[\frac{\epsilon}{\beta_t} + s_t^\theta(\alpha_t z + \beta_t \epsilon) \right]^2. \tag{41.41}$$

It would appear that in diffusion, one must train both the flow and the score, but for simple linear models the two can be related to one another, and one can choose the flow-matching or score-matching training procedure. For the example in Eq. 41.39, the corresponding score-matching training is

$$x_0 \sim p_0, \quad dx_t = \left[\left(\beta_t^2 \frac{\dot{\alpha}}{\alpha} - \beta_t \dot{\beta}_t + \frac{\sigma_t^2}{2} \right) s_t(x_t) + \frac{\dot{\alpha}}{\alpha} x_t \right] dt + \sigma_t dW_t. \tag{41.42}$$

One of the advantages of score- and flow-based methods is that one can reduce the architectural restrictions imposed by normalizing flows or autoregressive models. Score- and flow-based training avoid the normalization requirement. Score-based models learn gradients of log probability density functions on a large number of noise-perturbed data distributions, and then generate samples by Langevin-type sampling.

The generative models described in this subsection are called flow-based models [45], score-based generative models [46], diffusion probabilistic models [47], or stochastic interpolants [48]. They have several advantages over other model families. They often outperform GAN-level sample quality without adversarial training, and enable exact log-likelihood computation through their connection to continuous-time flows, which can be represented as a probability flow ordinary differential equation [47]. The main advantage is that the distribution $p(x)$ can be specified solely by its score or flow. This in turn enables more flexible model architectures than what can be used in normalizing flows or autoregressive models.

41.3.5 *Anomaly detection and out-of-distribution detection*

Unsupervised anomaly detection techniques detect anomalies in an unlabeled test data set under the assumption that the majority of the in-distribution data are normal under some measure, while out-of-distribution (OOD) data are not. In the context of autoencoders a popular technique is to use the reconstruction error of Eq. 41.22 to identify an outlier as one with a large reconstruction error [72–74]. One issue with this method is that for higher dimensional latent space and flexible neural network architectures the encoder-decoder map become identity for any input data, $f(x) = x$, regardless of whether input x is from the in-distribution training dataset or from the out-of-distribution data. The choice of autoencoder latent space dimensionality is thus an important hyperparameter that must be tuned.

Another set of anomaly detection techniques construct a model representing normal behavior from a given in-distribution training dataset, and then evaluate the likelihood of a test instance to be generated by the utilized model. For instance, one can use density estimation methods such as normalizing flows (section 41.3.4.3) to learn the density (likelihood) of the in-distribution training dataset $p(x)$, and apply it to the test data. The expectation is that out-of-distribution data will have a lower density (likelihood) under the in-distribution density model. This expectation is however not always met in high dimensions and the method suffers because likelihood-based training is sensitive to the smallest variance directions [75]. Low-variance directions may contain little or no information on the global structure of the image, so there is a mismatch between the training objective and outlier detection objective. Lower dimensional autoencoders with NF in the latent space deal better with this issue [54].

A related issue is that of typicality: an in-distribution data sample likelihood will typically be lower than the maximum value, so an out-of-distribution data sample that is closer to the peak would have a higher likelihood. If this happens in low-variance directions that dominate the likelihood, normalizing flows can assign higher likelihoods to out-of-distribution data than to in-distribution training data [76]. A number of techniques have been proposed to circumvent these limitations, such as likelihood regret [77], likelihood-ratio [75], likelihood in autoencoder latent space [54], and Wasserstein distance training of the likelihood $p(x)$ [68, 78]. These methods can achieve better anomaly detection performance than the autoencoder reconstruction error [53, 54, 78]. However, even perfect density estimation cannot guarantee good anomaly detection performance [79, 80].

Supervised anomaly detection techniques require a data set that has been labeled as in-distribution and out-of-distribution and involves training a classifier (the key difference to many other statistical classification problems is the inherent unbalanced nature of outlier detection). These methods assume some form for what out-of-distribution data may look like, and their success relies on whether the assumed form is a realistic representation of actual out-of-distribution data. When this assumption is valid these methods can be more powerful than unsupervised methods, but the reverse is also true. A hybrid between the two approaches is to train a classifier without labels [81]. All these approaches are largely complementary to each other [82]. Examples of different anomaly detection methods applied to HEP are the LHC Olympics 2020 and Dark Machines challenges [83, 84].

41.4 Self-supervised learning

Self-supervised learning (SSL) also aims to distill useful features in the data without requiring supervision labels for every sample in the input data. Self-supervised methods make use of large unlabeled datasets to build meaningful representations. They can generally be categorized as *autoassociative*, where the model is trained to reproduce or reconstruct its own (masked) input or *contrastive*, where the model is trained to learn a mapping that is insensitive to different "views" of the data. These methods are often used to build "foundation models" (FMs) discussed in

Sec. 41.9.11, which are pre-trained using self-supervised learning and fine-tuned using supervised learning for different downstream tasks. However, FMs are not the only possible use case.

A classic autoassociative task is masked language modeling popularized by the bidirectional encoder representations from transformers (BERT) model [85]. In this task, BERT ingests a sequence of words, a fraction of which are randomly masked, and tries to predict the original words that have been masked. For example, in the sentence "The Milky Way is a [MASK] galaxy," BERT would need to predict "spiral." This helps BERT learn bidirectional context. A variant of this approach is next token prediction, popularized by the generative pretrained transformer (GPT) [86]. A common theme in these methods is *tokenization*, in which elements of the input data are mapped to discrete vectors, known as tokens. These approaches have been applied in the context of particle jets [87–89], enabling the construction of backbone models that can be fine-tuned for different tasks and provide improvements for small training samples.

Sensory data (*e.g.*, 1D waveforms, 2D images, or 3D scenes) pose a significant challenge for autoassociative tasks compared to symbolic data such as language, math, and high-level physical concepts like jets and particles. For symbolic data, the masking unit is naturally defined (*e.g.*, a word for language) and associated with a strong semantic meaning, which yields a well-defined learning objective for mask-based self-supervision. On the contrary, sensory data captures raw information and a unit of data (*e.g.* a single pixel in an image) does not carry meaningful information alone. This challenge has resulted in in-depth R&D for self-supervision techniques in computer vision. The masked autoencoder (MAE) laid the initial ground work [90]: the authors discovered that a large fraction of masking (*i.e.*, 75%) is crucial for successful training using an asymmetric encoder-decoder architecture. Distillation with no labels (DINO) made another breakthrough by introducing a self-distillation technique where a student and teacher model pair—the teacher model typically being an exponential moving average of the student model—are forced to agree across different augmentations (*e.g.*, cropping, adding jitter, and rotating) of the same data instance [91, 92]. For effective representation learning of 3D geometrical shapes, multi-view projection matching techniques [93–95] are promising and a strong promise and relevant to time projection chamber (TPC) image data in high energy physics. Exploration of these specialized techniques in computer vision has impacted HEP applications [96, 97].

In contrastive learning, portions of the input data are paired together and the model is tasked to find matching pairs. Pairs can be constructed based on different data modalities, such as text and images, or based on data augmentations, that may be generic, such as adding noise, or domain-specific, like symmetry transformations. For example, the contrastive language-image pre-training (CLIP) [98] allows joint pretraining of a text encoder and an image encoder, such that a matching image-text pair have image encoding vector $\mathbf{z}_i$ and text encoding vector $\mathbf{z}_j$ that span a small angle, *i.e.*, have a large cosine similarity

$$c(\mathbf{z}_i, \mathbf{z}_j) = \frac{\mathbf{z}_i \cdot \mathbf{z}_j}{|\mathbf{z}_i||\mathbf{z}_j|} = \cos\theta_{ij} \ , \tag{41.43}$$

with θ_{ij} being the angle between the encoding vectors. This approach has been applied in astrophysics [99].

Positive pairs may also be constructed by applying data augmentations. For example, in the case of galaxy images, one may augment the data by performing image rotations, adding noise, size scaling, or adding point spread function smoothing, all of which are realistic transformations expected in a real galaxy image survey [100, 101]. For particle jets, tailored augmentations may include rotations about the jet axis, translations in the (η, ϕ) plane, smearing the positions of the soft jet constituents, and collinear splitting of the jet constituents [102–105]. Another augmentation strategy is based on re-simulating the stochastic shower and detector interactions, thus generating

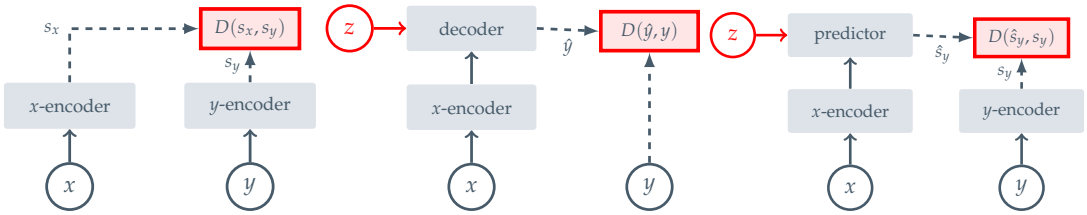

Figure 41.4: Common architectures for self-supervised learning, in which the system learns to assign a large scalar value to incompatible inputs, and a low scalar value to compatible inputs (M. Assran, et al. in ICCV, 2023). Joint-embedding architectures (left) learn to output similar embeddings for compatible inputs x, y and dissimilar embeddings for incompatible inputs. Generative architectures (center) learn to directly reconstruct a signal y from a compatible signal x, using a decoder network that is conditioned on additional (possibly latent) variables z to facilitate reconstruction. Joint-embedding predictive architectures (right) learn to predict the embeddings of a signal y from a compatible signal x, using a predictor network that is conditioned on additional (possibly latent) variables z to facilitate prediction.

multiple physical realizations of a primary particle's evolution [106, 107].

A well-known approach for contrastive learning with augmentations is SimCLR [108]. In this approach, the contrastive loss for a positive pair of an input and its augmentation $(\mathbf{z}_i, \mathbf{z}'_i)$ is defined in terms of the cosine similarity of Eq. 41.43 as

$$\mathcal{L}(\mathbf{z}, \mathbf{z}'_i) = -\ln \frac{\exp[c(\mathbf{z}_i, \mathbf{z}'_i)/\tau]}{\sum_{j \neq i \in \text{batch}} \Big[\exp[c(\mathbf{z}_i, \mathbf{z}_j)/\tau] + \exp[c(\mathbf{z}_i, \mathbf{z}'_j)/\tau]\Big]} \,, \tag{41.44}$$

and the total loss is given by the sum over all positive pairs in the batch, $\sum_{i \in \text{batch}} \mathcal{L}(\mathbf{z}_i, \mathbf{z}'_i)$. The loss decreases when the distance between positive pairs decreases or when the distance between negative pairs increases. The hyperparameter τ is known as temperature and controls the relative influence of positive and negative pairs. SimCLR has been applied in radio astronomy [109], neutrino physics [110], and collider physics [102]. Another application of contrastive regularization is self-distillation introduced in DINO discussed above. Self-distillation is a powerful technique that can be applied regardless of the target task, and improves the quality of self-supervision for many computer vision models for both image and point cloud data.

Finally, an alternative paradigm is the joint-embedding predictive architecture (JEPA) [111], which learns meaningful representations by modeling missing or unseen embeddings directly in the latent space without a decoder or full input reconstruction. The advantages of this approach are no data augmentations are required and unnecessary details of the input can be ignored. This approach has been applied to particle jets [112, 113] and Square Kilometer Array (SKA) light cones [114]. A comparison between the different self-supervised learning approaches can be found in Fig. 41.4, reproduced from Ref. [111].

41.5 Optimal control, reinforcement learning, and active learning

Many problems in science and engineering can be cast as a control problem, which comprises a cost functional that is a function of state and some control variables that specify some underlying dynamical system. This is relevant for the control of accelerators where the dynamical system is physical. This formalism can also be used to describe the design of experiments, planning of an observational survey, and other decision making processes relevant to the scientific method. It is

closely connected to planning, dynamic programming, and reinforcement learning. Optimal control generalizes the framing of learning presented in Sec. 41.2.1.

41.5.1 *Optimal control*

Optimal control theory deals with finding a control for a dynamical system over a period of time such that the objective function is optimized. The underlying system can be discrete or continuous and may be deterministic or stochastic. The commonalities and differences between optimal control and reinforcement learning can be best understood through the formalism of a Markov decision process (MDP), which is a discrete-time stochastic control process.

A Markov decision process comprises four components often organized as a 4-tuple (S, A, P_a, R_a), where: S is a set of states called the state space, A is a set of actions called the action space, $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t+1$, $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a.

The policy function π is a mapping from state space to action space that can be either deterministic or probabilistic. For examples, the policy that drives a computer chess playing system, decides which move to make given the current state of the board. Similarly, policies dictate which experiment should be built next, which field of the sky should be observed, or how to adjust the operational parameters of an accelerator. The dynamics of the resulting system are then fixed by combining the policy with the underlying MDP. The evolution of the resulting dynamical system behaves like a Markov chain since the action chosen in state s is completely determined by $\pi(s)$ and $\Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ implies the Markov transition matrix $\Pr(s_{t+1} = s' \mid s_t = s)$.

The objective optimal control is to choose a policy π that will maximize a cumulative function of the instantaneous rewards R_a. A common choice is the expected discounted sum:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\right], \tag{41.45}$$

where $a_t \sim \pi(s_t)$ are the actions given by the policy, the expectation computed with respect to the distribution $s_{t+1} \sim P_{a_t}(s_t, s_{t+1})$, and γ is the discount factor satisfying $0 \leq \gamma \leq 1$. The discount factor is usually close to 1 and sometimes reparameterized as $\gamma = 1/(1+r)$, where r is called the discount rate. A lower discount factor motivates the decision maker to favor taking actions early, rather than postpone them indefinitely.

A policy that maximizes the objective function is called an optimal policy and denoted π^*, though the optimal policy need not be unique. Importantly, the Markov property implies that the optimal policy is only a function of the current state. Dynamic programming can be used to find the optimal policy for MDPs with finite state and action spaces. For instance, in value iteration (a.k.a. backward induction) can be used to solve the "Bellman equation" [115]. For continuous-time systems, the optimal policy is defined by the Hamilton–Jacobi–Bellman equation [116].

In many settings, it is assumed that the state s is fully known when action is to be taken and there are no latent variables. When this assumption is not true, the problem is called a partially observable MDP. These problems are generally more difficult and the dynamic programming algorithms do not directly apply [117].

41.5.2 *Reinforcement learning*

The main difference between the classical dynamic programming methods and reinforcement learning (RL) algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible. For example, RL was used in the context of jet physics to search for the most likely jet clustering when the number

of constituents was too large for the exact dynamic programming algorithm to be used [118]. In addition, RL can be used when the probabilities or rewards are unknown. Instead, the transition probabilities are often accessed indirectly through interaction with a real or simulated environment.

Numerous variations to RL exist, which include so-called model-based and model-free approaches (referring to models of the instantaneous rewards and the state transitions) and on-policy and off-policy (which describes how the actions taken during learning are related to the current policy). See Ref. [119] for an introduction and Ref. [120] for a recent review. Some examples of RL use in particle physics are in Refs. [121–123].

41.5.3 *Multi-arm bandits*

Multi-arm bandit problems are a classic reinforcement learning problem where one tries to maximize the expected gain by allocating a limited set of resources to various alternatives. The name is a reference to a gambler with a fixed amount of money that must choose between multiple slot machines (or "one-armed" bandits) when the payoff for the individual machines is unknown. A hallmark of multi-arm bandit problems is that they involve a tradeoff between exploration (playing machines to estimate their payoff) and exploitation (playing machines with the highest estimated payoff). Multi-armed bandits are used to manage large projects, organizations, and scheduling problems. The theory has a long history going back to Robbins in 1952 that used it to study the sequential design of experiments [124] and Gittins who derived an optimal policy under some conditions [125].

41.5.4 *Bayesian optimization*

A closely related set of techniques involve optimizing some expensive black box function $f(x)$. For instance, the function may be computationally expensive to evaluate or low-latency, *e.g.* it may involve manually re-configuring a system. This is particularly relevant for analysis optimization in particle physics where evaluating $f(x)$ involves processing large numbers of simulated collisions. Another common use case involves optimizing the hyperparameters of a learning algorithm.

Without any assumptions about the function $f(x)$ this is hopeless; however, if one assumes something about the functions (*e.g.* some notion of smoothness) then one can leverage function evaluations evaluations $\{f(x_t)\}_{t=1,\ldots,T}$ to say something about what value the function might take on at other values of x. This is usually cast in Bayesian terms, and Gaussian processes (Section 41.8.2) are often used to model the distribution over $f(x)$. The optimization techniques that use this framing are generically referred to as Bayesian optimization [126].

Optimization in this context is usually characterized by an *exploration-exploitation* tradeoff, similar to what is found in multi-arm bandits. Here, exploration refers to function evaluations that characterize the function in regions that haven't been evaluated, while exploitation refers to evaluations near what is predicted to be its maximum. This setting is similar to reinforcement learning in that it involves sequential decisions (*i.e.*, where to evaluate the function next), but usually the target function $f(x)$ is assumed to be static. In that sense, the state referred to in the language of an MDP is the state of knowledge about the function after sequential evaluations $\{f(x_t)\}_{t=1,\ldots,T}$. The reward at time t is not the value of the function $f(x_t)$, but some quantity that characterizes what was learned about the function's maximum. In this literature, one often refers to the *acquisition function*, which plays a similar role as the expected value of the reward in RL. Common acquisition functions include the probability of improvement, the expected improvement, and an upper-confidence bound [127].

41.5.5 *Active learning*

Active learning is closely related to Bayesian optimization, described above. In Bayesian optimization one estimates the function $f(x)$ from some set of evaluations $\{y_t = f(x_t)\}_{t=1,\ldots,T}$; however,

the goal is to find the maximum $x^* = \arg\max_x f(x)$. In active learning, the goal is not to find the maximum of $f(x)$, but to approximate the function as one does in supervised learning. The main difference compared to vanilla supervised learning is that the labeled training dataset isn't provided a priori in a passive way, but the learning algorithm actively decides where to generate $(x_t, y_t = f(x_t))$ pairs. The function $f(x)$ is sometimes referred to as an *oracle*. Active learning is particularly attractive when obtaining labeled data is a costly process.

More broadly, a challenge of many machine learning applications is obtaining labeled data, which can be a costly process. If a system could learn from small amounts of data, and choose by itself what data it would like the user to label via an external process called oracle, it would make machine learning more powerful. Such frameworks are also called experiment design or active learning. In active learning, a model is trained on a small amount of data (the initial training dataset), and an acquisition function (often based on the model's uncertainty) decides on which data points to ask for a label. The acquisition function selects one or more points from a pool of unlabeled data points, with the pool points lying outside of the training dataset. Once we label the selected data points, these are added to the training dataset, and a new model is trained on the updated training dataset. This process is then repeated, with the training dataset increasing in size over time. The advantage of such systems is that they often result in dramatic reductions in the amount of labeling required to train an ML system (and therefore cost and time).

41.6 Simulation-based inference

The goal of simulation-based inference (related to, but distinct from, likelihood-free inference) is to extend the statistical procedures described in the Chapter on Statistics (*e.g.* parameter estimation, hypothesis tests, confidence intervals, and Bayesian posterior distributions) to the situation where one does not know the explicit likelihood $p(x|\theta)$, the probability of the data given the parameters θ, but has access to a simulator that defines the likelihood $p(x|\theta)$ implicitly [128, 129]. In a typical setup we would like to solve the so called inverse problem of getting the posterior of the parameters given the data, $p(\theta|x)$, but we cannot use Bayes theorem directly because we do not have explicit $p(x|\theta)$.

In particle physics and cosmology, the simulators usually use Monte Carlo event generators (see Sec. 43) to sample unobserved latent variables z, such as the z_p phase space of the hard scattering (see Sec. 49.4), z_s associated to showering and hadronization, z_d associated to the interaction of particles with the detector (see Sec. 34), or initial Gaussian modes of the universe realization. As such, the full simulation chain can be expressed approximately as

$$p(x|\theta) = \int dz p(x, z|\theta) = \int \mathrm{d}z_d \int \mathrm{d}z_s \int \mathrm{d}z_p \, p(x|z_d)p(z_d|z_s)p(z_s|z_p)p(z_p|\theta) \,, \tag{41.46}$$

where θ are the Lagrangian parameters that dictate the hard scattering. Evaluating the marginal likelihood $p(x|\theta)$ is intractable as it would require evaluating the integral above for each event.

While the marginal likelihood is intractable, simulators provide the ability to generate synthetic data $x_i \overset{\text{i.i.d.}}{\sim} p(x|\theta)$ for any value of the parameters θ. One can use a suitable proposal distribution $\tilde{p}(\theta)$, sample $\theta_i \overset{\text{i.i.d.}}{\sim} \tilde{p}(\theta)$, generate synthetic data $x_i \sim p(x|\theta_i)$, and then assemble a training dataset $\{x_i, \theta_i\}_{i=1,\ldots,n}$ that can be used to train various machine learning models.

There is thus a close analogy between simulation-based inference and data driven machine learning tasks discussed so far, replacing θ with y. One difference is that in simulation-based inference we can always generate new samples by running additional simulations, while we typically view training dataset in machine learning as fixed. This property of simulation-based inference enables active learning, where the additional simulations are chosen such as to minimize the error on the desired statistical inference task. Another difference is that we often have access to the joint

likelihood $p(x, z|\theta)$, where z are unobserved latent variables[3].

Typically in particle physics, one uses histograms or kernel density estimation to model the distribution of observables (low-dimensional summary statistics such as the invariant mass) of simulated data [130]. Alternatively, one can use an explicit parametric family (such as a falling exponential or a Gaussian distribution) to model $\hat{f}(x|\theta) \approx p(x|\theta)$. That model is then used as as a surrogate for the unknown density implicitly defined by the simulator. A related approach is known as approximate Bayesian computation (ABC), which approximates the likelihood through an acceptance probability that synthetic data is sufficiently close to the observed data [131, 132]. In practice, these techniques are limited to low-dimensional representations of the data. Thus the potential of recent machine learning approaches to simulation-based inference is to extend this approach to higher-dimensional data, while maintaining the already well-established statistical procedures.

For instance, one can use normalizing flows (see Sec. 41.3.4.3) and the loss functions for density estimation (see Sec. 41.3.3) to learn a surrogate model for the likelihood $\hat{f}(x|\theta) \approx p(x|\theta)$ [69]. Similarly, one can use conditional density estimation to learn a surrogate model for the posterior $\hat{f}(\theta|x) \approx p(\theta|x)$, which may involve including the prior-to-proposal ratio $\tilde{p}(\theta)/p(\theta)$ [70]. In addition to the unsupervised learning techniques, one can also use supervised learning to learn the likelihood-ratio $r(x|\theta_0, \theta_1) = p(x|\theta_0)/p(x|\theta_1)$ by leveraging the *likelihood-ratio trick* of Eq. 41.13 [133, 134].

In some cases one can also augment the training dataset to include the joint likelihood-ratio

$$r(x_i, z_i|\theta_0, \theta_1) \coloneqq p(x_i, z_i|\theta_0)/p(x_i, z_i|\theta_1) \,, \tag{41.47}$$

which can be used to reduce the variance for the squared-error or cross-entropy losses [134, 135]. While the marginal likelihood $p(x|\theta)$ is intractable due to the high-dimensional integral over the latent space, the joint likelihood is often tractable since no integration is necessary.

In some cases performing the marginal integral of Eq. 41.46 is tractable even for high dimensional latent space z. One of the approaches to make it feasible in high dimensional latent space is to make simulations differentiable with respect to all of its parameters, global variables θ and latent variables z. While differentiable simulations have not traditionally been developed for scientific applications, the success of machine learning based on backpropagation combined with gradient descent (see Sec. 41.9.1), has inspired a renewed interest. One example is FlowPM cosmological N-body simulation, which takes advantage of Mesh-Tensorflow to achieve a GPU-accelerated, distributed, and differentiable simulation [136]. Availability of simulation gradients in turn enables gradient based Monte Carlo Markov chain methods to perform high dimensional marginal integral over the latent space z and over parameter space θ [137].

Often SBI uses predetermined summary statistics, such as binned histograms in particle physics, or power spectrum in cosmology, to avoid the curse of dimensionality. It is however possible to train on uncompressed high dimensional data in cosmology by exploiting the symmetries [138]. Yet another alternative is to train the network to search for the best possible summary statistic. The summary statistic can then simply be $\hat{\theta}$, which is the estimate of the parameters θ that emerge from a supervised training on simulations. In SBI these can often be biased even after training, and one possible solution is to form a pseudo-likelihood to model the bias as a function of the true value of θ [139].

41.6.1 *Latent space reconstruction and unfolding*

While much of the work on simulation-based inference described above is aimed at inferring the parameters θ of the simulator, there is work that aims to infer the latent variables z. A common

[3] For this reason we prefer to use simulation-based inference instead of likelihood-free inference: joint likelihood $p(x, z|\theta)$ is often available, it is the marginal integral over latent space z that is assumed to be intractable.

approach in particle physics is to think of the parameters θ as parameters of a theory, such as masses, coupling constants, or Lagrangian parameters, while z might describe the kinematics of a collision before the detector response. Inferring the distribution $p(z|\{x_1, \ldots, x_n\})$ from a dataset of multiple observations is commonly referred to as unfolding in particle physics, and deconvolution in other contexts. Unfolding is a classic inverse problem, and the collection of ideas being used for machine-learning based simulation-based inference are also being applied in this setting [140]. For example, the OmniFold method [141] iteratively reweights a dataset in an unbinned way using machine learning to produce a simultaneous measurement of many observables. In this method, samples $\vec{x}_r$ from detector-level MC simulation are first corrected by a learned weighting function $\omega(\vec{x}_r)$ to match data. Then, samples $\vec{x}_p$ from particle-level MC simulation are corrected by another learned weighting function $\nu(\vec{x}_p)$ to match the $\omega(\vec{x}_r)$-weighted MC simulation. The method is iterated multiple times, to achieve $\nu(\vec{x}_p)$-weighted MC events whose event yields and kinematics match those observed in data. The H1 [142] and ATLAS [143] Collaborations have used the OmniFold method in experimental measurements. It has also been applied to T2K [144] and CMS [145] open data.

In cosmology, a common task is to reconstruct initial density distribution of the dark matter, or its final distribution, from data such as galaxy positions. This can then be used for various downstream tasks such as cosmological parameter inference or making maps of dark matter in our universe. High dimensional SBI can be used for this task [146]. An alternative is Bayesian inverse problem inference using the forward model $g(z, \theta)$, which can be an N-body simulation with some simple galaxy formation model added to it [137, 147, 148]. Standard Bayesian methodology using for example MCMC can be used to solve this task and find the posterior $p(z, \theta|x)$, which specifies initial distribution of dark matter. To draw samples of final dark matter distribution, and of the reconstructed data, we can first draw samples from the posterior $p(z, \theta|x)$, and then evaluate forward model $g(z, \theta)$ for each sample.

41.7 Data representations, inductive bias, and example applications

In Sec. 41.2 we describe the input data as living in an abstract space $x_i \in \mathcal{X}$. In this section, we briefly discuss some of the common types of structured data that are encountered in physics and refer to the corresponding models classes that have been developed to work with them. We elaborate on the model classes in more detail in the following section.

The most basic and common type of data structure is when $\mathcal{X} = \mathbb{R}^d$. This is often referred to as *tabular data* since the entire data set $\{x_i\}_{i=1,\ldots,n}$ can be thought of as a table with n rows and d columns. It is common to think of an individual entry x_i as a vector in d-dimensional Euclidean space, where the coordinates correspond to the columns of this table. In some cases individual components of x_i might be integers or take on only discrete values, in which case describing the space of the data as real-valued is a slight abuse of notation and representation. For many years this was the dominant type of data in high energy physics as it is a natural input type for shallow neural networks, multilayer perceptrons, support vector machines, and tree-based methods found in popular tools such as `TMVA` [149].

For categorical data, one typically uses a numerical representation such as *integer encoding* where different categories are mapped to integers with a corresponding dictionary. Another common representation of categorical data is based on the so-called *one-hot encoding* (aka 'one-of-K' or 'dummy'), in which case the category is mapped to a k-dimensional binary vector where k is the number of categories and each component of this vector corresponds to a particular category. In the one-hot encoding, only one of the components is non-zero. Finally, there are approaches in one learns an *embedding* that maps discrete categories into $\mathbb{R}^d$; an example of this is `Word2Vec` [150]. Interestingly, such embeddings can preserve various types of semantics; for instance, the vector

`walking - walk` is similar to the vector `swimming - swam` as are the vectors connecting countries and their capital cities. This allows for a loose sense of algebra on the word embeddings such as `walking - swimming + swam = walk`. Similar types of embeddings have also been used in a number of scientific use-cases including biological sequences (*e.g.*, DNA, RNA, and proteins) for bioinformatics applications [151].

Particle physics data often is represented with an extension of the simple tabular data structure where the number of columns is not fixed. For instance, if the rows correspond to data for individual collisions, the number of electrons (and positrons) reconstructed in the event is variable. Thus the number of columns needed to represent the energy, momentum, and charge of these particles is also variable. A common solution to this problem is to fix a maximum number of particles and then *truncate* and *zero-pad* to fit the data into a fixed tabular representation, though this is not the natural representation of the data and it leads to a loss of information.

Sequential data is also commonly encountered in physics (*e.g.* in time series). Here an individual entry $x_i = (x_i^1, \ldots, x_i^t, \ldots x_i^{T_i})$ where t is index for the ordered sequence, T_i is the length of the sequence (which might be variable), and the data associated to each "time" $x_i^t \in \mathbb{R}^d$. This is similar to the previous example where the energy, momentum, and charge of the tth electron in the ith event would be x_i^t and the electrons might be sorted according to their energy or transverse momentum. Sequential data is also encountered in natural language processing, where x_i^t correspond to individual words in a sentence. Recurrent neural networks (see Sec. 41.8.4.5) are particularly well suited to sequential data. Examples applications from the Living Review include Refs. [152–157].

Image-like data is one of the most dominant forms of data in industrial applications of deep learning, is very relevant for astronomy and cosmology, and also appears in particle physics in various forms. Image-like data typically involves d-dimensional features associated to a regular grid or lattice that does not vary across the individual instances x_i. The canonical example is a standard image from a camera with $W \times H$ pixels where the pth pixel has data $x_i^p \in \mathbb{R}^3$ corresponding to the three *channels* in the RGB color model. It is important to recognize that the data corresponding to the 2-dimensional image is not 2-dimensional; instead, it is $(W \times H \times c)$-dimensional, where c is the number of channels. In astronomy, an image may be grey scale ($c = 1$) or there may be more *channels* ($c > 3$) corresponding to different color filters. In other applications, the grid or lattice might be 3- or 4-dimensional. For example, the data associated to a regularly segmented calorimeter can be thought of as a 3-dimensional image and the data associated to a lattice simulation of a classical or quantum system can be thought of as a 4-dimensional image. Convolutional neural networks, described in Sec. 41.8.4.4, are particularly well suited to image-like data. Example applications from the Living Review include Refs. [153, 158–180].

It is also possible that the data (or features) associated to one "pixel" or lattice site may itself be structured. For example, the single read-out plane of a liquid argon time projection chamber (LArTPC) may involve a 1-dimensional or 2-dimensional grid, but the data associated to each "pixel" is itself a sequence or waveform. Example applications in neutrino physics from the Living Review include Refs. [26, 27, 30, 181–208]. Similarly, in lattice quantum chromodynamics, the data associate to a particular site (or link) would be group valued (*e.g.* $x_i^p \in SU(3)$ as in Refs. [209, 210]).

Both sequential and image-like data have a notion of temporal or spatial structure. While it is possible to unroll an image into a $(W \times H \times c)$-dimensional vector, that would erase the spatial structure and obfuscate the fact that nearby pixels are highly correlated. Similarly, one could permute the time index for sequential data, but that would destroy the temporal structure of the data. The complementary point of view is that the model class should also be aware of the structure of the data. Recurrent and convolutional neural networks are good examples of *inductive bias* as the models incorporate the structure of the data. In some cases this can be formalized in terms of symmetry. For example, if we train model to classify images of cats and dogs, we would like it's

prediction to be invariant to where in the image the cat is. This type of translational invariance can be enforced in the design of the model class.

While permuting the elements of a sequence destroys the temporal structure of a time series, attaching a temporal index t to a set of objects with features x_i^t can also be problematic. If the data corresponding to x_i are really a set $\{x_i^1, \ldots, x_i^{T_i}\}$ (*e.g.*, a point cloud), then we would like the output of the model to be *permutation invariant* or *permutation equivariant* depending on if the output is per-set or per-element, respectively. A standard sequential or convolutional model will not generally be permutation invariant, but models such as deep sets, various types of graph neural networks, and transformers can be made to enforce permutation symmetry. Example applications from the Living Review include Refs. [172, 211–222].

The temporal and spatial structure of sequences and image like data can also be generalized. For instance, a 1-dimensional sequence can be generalized to a tree structured data like one finds in the hierarchical clustering of jets or as in a directed-acyclic graph (DAG). Generalizations of recurrent neural networks have been constructed that can operate over these more complex data structures [223, 224]. More generally, one can considered graph-structured data composed of nodes and edges or multi-graphs that group together three nodes into faces or k nodes into k-edges. Graph neural networks are a class of models that work with this type of data. The emerging subfield of geometric deep learning aims to unify the notation, terminology, and theory that connect these considerations of the structure of the data and the corresponding model architecture. Example applications in the Living Review include Refs. [32, 190, 197, 200, 201, 203, 225–250].

If the data are expected to have a symmetry associated to them but one is working with a model class that does not enforce this symmetry, then *data augmentation* is a common procedure used to improve generalization performance. Here one starts with an initial dataset $\{x_i\}_{i=1,\ldots,n}$ and produces an augmented dataset $\{x'_i\}_{i'=1,\ldots,n'}$ through some data augmentation strategy. For example, one might apply a random rotation $R_{i'}$ to an image to produce $x'_i = R_{i'}(x_i)$ if one assumes rotational invariance in the underlying problem.

In some cases some of the individual features (components) of x are functions of other features. For instance, one may include components of a four-vector (E, p_x, p_y, p_z) as well as redundant information such as transverse momentum, azimuthal angles, rapidity, etc. In this case, the data is restricted to a lower-dimensional surface embedded in $\mathcal{X}$. Even if the features aren't redundant, statistically the data are often effectively restricted to a small subspace of statistically likely samples and those that are exceedingly unlikely. For instance, the space of natural images is a small and highly structured subspace of all possible images, which are dominated by what we would perceive visually as noise. The term *data manifold* is used to describe this restricted subspace where the data are to be found, even though it does not necessarily satisfy the formal requirements of a manifold in the mathematical sense.

These considerations on the structure of the data not only apply not to the input data $x_i \in \mathcal{X}$, but also to the output data $y_i \in \mathcal{Y}$. For instance, one might want a sequence-to-sequence model as in machine translation of written text [251] or to learn a function that takes sets as input and produces graphs as output as in the Set2Graph mode [252]. One might also want the input and output of the model to be different in representations of an underlying symmetry group and for the model to enforce group-equivariance [209, 210]. The development of the necessary modeling components to enable practitioners to compose and train these types of models is a significant development for the field of physics.

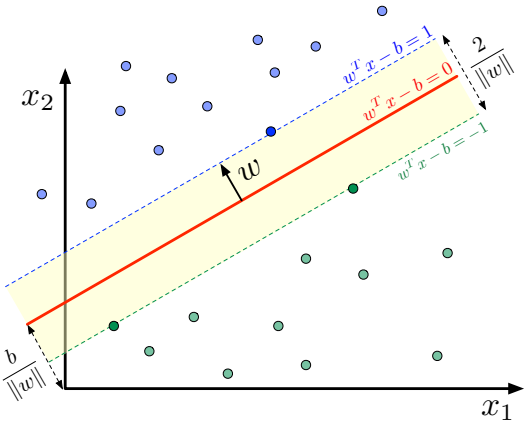

Figure 41.5: Illustration of a maximum margin classifier for a linear support vector machine in the separable case.

41.8 Flavors of ML models

41.8.1 *Support vector machines*

Support vector machines (SVMs) are a class of supervised learning models used for classification and regression. The learning algorithm involves a convex optimization problem that has a unique solution and can be solved with quadratic programming techniques. In this sense, they are robust and easier to characterize than neural networks that involve non-convex optimization.

Linear support vector machines are used for binary classification, where $\mathcal{X} = \mathbb{R}^d$ and the target labels are conventionally defined as $\mathcal{Y} = \{-1, 1\}$. The classification is simply based on which side of a hyperplane the data lie. Any hyperplane can be written as the set of points x satisfying $w^T x - b = 0$, where $w, b \in \mathbb{R}^d$ are the parameters of the model. The vector w is normal to the hyperplane, but not necessarily normalized. The quantity $\frac{b}{\|w\|}$ quantifies the offset of the hyperplane from the origin along the normal vector w.

If the training dataset is linearly separable, then there is a region bounded by two parallel hyperplanes, called the *margin*, that separate the two classes of data. The maximum margin classifier is uniquely defined by making the distance between these two hyperplanes as large as possible. The boundaries of the margin can be defined by $w^T x_i - b = \pm 1$, and the width of the margin is given by $\frac{2}{\|w\|}$. Figure 41.5 illustrates this for $x \in \mathbb{R}^2$.

Since the width of the margin is maximized when $\|w\|$ is minimized, we can state the goal of the (hard) maximum-margin classifier in the linear separable case as the following constrained optimization problem: Minimize $\|w\|^2$ subject to the constraint $y_i(w^T x_i - b) \geq 1$ for $i = 1, \ldots, n$. The w and b that solve this problem uniquely determine the resulting classifier, $\hat{y}(x) = \text{sgn}(w^T x - b)$. This geometric description makes it clear that the maximum-margin hyperplane is completely determined by those x_i that lie nearest to it: the eponymous *support vectors*.

41.8.2 *From Bayesian linear regression to kernel regression and Gaussian processes*

As discussed in Sec. 41.2.2, linear regression is a specific case of regression where the solution is parameterized as a linear combination of basis functions $\phi(x)$,

$$f_\phi(x) = \sum_k w_k \phi_k(x) = w^\intercal \phi, \tag{41.48}$$

using a short-hand vector notation. If we aggregate all the basis functions of the training data into $\Phi(x)$ and all x into X, and assuming a Gaussian noise model $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$, we can write the noise

probability distribution as

$$p(y|X,w) = \mathcal{N}(w^\intercal \Phi, \sigma_n^2 I). \tag{41.49}$$

In overparametrized models, this needs to be regularized, with explicit L2 norm of the weights, as discussed in Sec. 41.2.5. If we view the process in the Bayesian context, we add a weight prior $p(w) = \mathcal{N}(0, \Sigma_p)$ to the data likelihood. With this one can define the posterior of the weights as

$$p(w|X,y) \propto p(y|X,w)p(w) = \mathcal{N}(\sigma_n^{-2} A^{-1} \Phi y, A^{-1}), \tag{41.50}$$

where $A = \sigma_n^{-2} \Phi \Phi^\intercal + \Sigma_p^{-1}$.

Our main task is not to predict the weights themselves, but to predict f^* given some x^*. In the Bayesian view, one must model average over the weights,

$$p(f^*|x^*,X,y) = \int \mathrm{d}w p(f^*|x^*,w)p(w|X,y) = \mathcal{N}(\sigma_n^{-2}\phi(x^*)^\intercal A^{-1}\Phi y, \phi(x^*)^T A^{-1}\phi(x^*)), \tag{41.51}$$

where we used $p(f^*|x^*,w) = N(0,\sigma_n^2)$. This can be rewritten as

$$p(f^*|x^*,X,y) = \mathcal{N}(\phi(x^*)^T \Sigma_p \Phi (K + \sigma_n^2 I)^{-1} y, \phi(x^*)^\intercal \Sigma_p \phi(x^*)) - \phi(x^*)\Sigma_p \Phi (K+\sigma_n^2 I)^{-1} \Phi^T \Sigma_p \phi(x^*)), \tag{41.52}$$

where $K = \Phi^T \Sigma_p \Phi$. In general we call $k(x,x') = \phi^T(x)\Sigma_p \phi(x')$ a kernel or covariance function between x and x'. The final expression for the regression mean and covariance has a form

$$f^* = K(x^*, X)[K(X,X) + \sigma_n I]^{-1} y, \tag{41.53}$$

with covariance

$$\mathrm{cov}(f^*) = K(x^*,x^*) - K(x^*,X)[K(X,X) + \sigma_n^2 I]^{-1} K(x^*,X). \tag{41.54}$$

One can see that the prediction takes the observed values y at the training data X, and predicts the value at x^* by incorporating the strength of their correlations via $K(x^*,X)$. In addition, the prediction also suppresses the training points with a small inverse covariance, a generalization of inverse noise weighting.

This expression simply rewrites the standard Bayesian regression, and the kernel is still defined as an inner product of the regression basis functions $\phi(x)$ with respect to Σ_p. However, the kernel can be replaced by any kernel form that describes the level of correlations between x and x', a process known as the *kernel trick*. A general condition for the kernel to be valid is that the covariance matrix is always invertible. In a Gaussian process the kernel is often stationary, defined as $k(x,x') = f(|x-x'|)$. This and many other kernels cannot be related to a finite set of basis functions $\phi(x)$, which is why it is often stated that Gaussian process corresponds to an infinite basis function expansion. Yet another form of the kernels are neural tangent kernels of neural networks in the infinitely wide network limit [23].

Another path to Gaussian processes is via kernel regression, where one makes kernel regression Bayesian [253]. Standard kernel regression, also called Nadaraya-Watson regression, is of the form $f^* = \sum_x k(x,x^*)y(x) / \sum_x k(x,x^*)$, which can be interpreted as a soft version of the k-nearest neighbors algorithm. Bayesian kernel regression in the form of a Gaussian process replaces kernel sums with matrix operations, and is closer to linear regression than to the nearest neighbor methods.

One advantage of Gaussian process is that one can work with a family of kernel functions parameterized by some hyperparameters η. One can then optimize the hyperparameters via gradient-ascent on the marginal likelihood function. In contrast, hyperparameter tuning in other models

typically requires a grid search or some other black-box optimization procedure evaluated on held out data or some form of cross-validation.

While we only describe Gaussian process regression, there is a corresponding Gaussian process classification. Rasmussen and Williams provides an excellent review of Gaussian processes [254]. Numerically, Gaussian process libraries are confronted with computing the inverse of the covariance kernel, which scales like $\mathcal{O}(n^3)$ in computational complexity. Gaussian processes are often used as emulators or surrogate models, specially in the context of low dimensional input x and low number of training data n to avoid the steep $\mathcal{O}(n^3)$ scaling. They are used widely in cosmology, and there are a growing number of applications in (astro-)particle physics [255]. Recent works explore the design of physics-inspired kernels and use Gaussian processes to model the intensity for a Poisson point process like those found in experimental particle physics and γ-ray and X-ray astronomy [256–258]. Gaussian processes are also extensively used in Bayesian optimization (Section 41.5), because the uncertainty quantification that is automatically provided by the Gaussian process enables exploration-exploitation strategies where to evaluate the function next.

41.8.3 *Decision trees*

Tree-based models Classification and regression trees (CART) typically partition the input space into J disjoint regions $\mathcal{X} = \mathcal{X}^1 \cup \cdots \cup \mathcal{X}^J$ through a sequence of $J-1$ binary splits based on an individual components of $x \in \mathcal{X}$ (*e.g.* $x_4 < 0.7$) [259]. The model is piecewise constant and assigns the value $b_j \in \mathcal{Y}$ to the jth terminal region $\mathcal{X}^j$. The model can be written

$$\hat{y}(x) = f_\phi(x) = \sum_j b_j \mathbf{1}(x \in \mathcal{X}^j) \,. \tag{41.55}$$

The parameters ϕ of the model comprise the components index and thresholds for the successive splittings and the coefficients b_j.

Tree learning refers to the algorithm used to choosing the tree structure and determining the predictions at leaf nodes. Optimization of the tree structure involves a difficult discrete optimization since the change in the loss with respect to the tree structure is non-differentiable and it is intractable to explore the combinatorially large space of possible trees with brute force. Therefore, the discrete optimization component of tree learning typically involves some approximate algorithm based on heuristics. In contrast, optimization of the b_j for a given tree structure can exploit gradient-based optimization algorithms.

Common approaches to building the decision tree start with a root node and grow with splits based on individual attributes (components of x). These are referred to as top-down induction strategies. There are various impurity heuristics used for choosing the best attribute to split on such as the Gini index, cross-entropy and mis-classification error. Generally they aim to find a split that will refine the the terminal nodes such that they have higher purity than the parent node.

Because most tree learning algorithms consider splits aligned with individual feature components, there are some failure modes for tree-based models. However, tree-based models work well with tabular data composed of a mix of continuous and discrete features. Tools such as `XGBoost` [260] and LightGBM [261] are competitive on tabular data benchmarks like TabArena [262] and are widely used in industry; the boosted decision trees (BDTs) implemented in `StatPatternRecognition` [263] and `TMVA` [149] have been one of the most used techniques in particle physics [3].

Individual trees are often referred to as weak learners and they can be combined in various ways described below. Regularization is also an important consideration with tree-based models as one can always learn a tree that assigns exactly one training dataset point per terminal node and memorize the training dataset exactly. One approach to this is called *pre-pruning*, which simply

terminates the growing of the trees if the number of training samples reaching the terminal node drops below some threshold, the purity of a terminal nodes is below some threshold, or if the improvement in purity due to a proposed split is not above a threshold. Another regularization approach is called *post-pruning*, which uses a validation data set that is disjoint from the training dataset to probe generalization performance. In this approach, after initially growing a tree with the training dataset, a sequence of pruned trees is considered where splits are removed based on some heuristic. The tree in this sequence of pruned trees that minimizes the generalization error on the validation set is chosen. Alternatively, in tools such as `XGBoost` there is an explicit regularization term included in the loss function (see Eq. 41.62).

Ensemble methods The idea of ensemble methods is to combine multiple models into a more performant one by exploiting the bias-variance tradeoff [264]. This is most commonly achieved through averaging (*e.g.* bagging and random forests), which primarily reduces variance, or boosting (*e.g.*, AdaBoost and gradient boosting), which primarily reduces bias. Here, bias refers to the difference between the Bayes optimal model and the average model produced by the learning procedure with different training sets and variance quantifies how much the learned model varies from one training set to another.

The motivation of boosting is to combine the outputs of many "weak" models to produce a more expressive model. Compared to averaging techniques like bagging and random forests, the model is built sequentially on modified versions of the data and the final predictions are combined through a weighted sum

$$\hat{y}(x) = \sum_{t=1}^{T} \beta_t \hat{y}_t(x) \ , \tag{41.56}$$

where β_t expand the parameters of the model ϕ.

Bagging The idea behind bagging (bootstrap aggregation) is to create T bootstrap training datasets $B_1, \ldots, B_T$ drawn from the training dataset $\{x_i, y_i\}_{i=1,\ldots,n}$, then learn a model $\hat{y}_t$ for each, and finally construct an average model $\hat{y}(x) = (1/T) \sum_t \hat{y}_t(x)$. If one had T independent training datasets each of size n, 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. By using bootstrap resampling, the bias may increase but the reduction in variance often dominates, which leads to improved performance.

Random forests Random forests refers to a type of "perturb and combine algorithm" that combines bagging and random attribute subset selection. Again one builds trees $\hat{y}_t(x)$ from bootstrap training datasets B_t, but instead of choosing the best split among all attributes, one select the best split among a random subset of k attributes. If k includes all attributes, then it is equivalent to bagging.

AdaBoost In AdaBoost (adaptive boost) the sequence of trees $\hat{y}_1, \ldots, \hat{y}_T$ are trained with reweighted versions of the original training dataset such that the weight of individual training sample is based on the prediction error in the previous iteration [265]. This requires working with a loss function that and learning procedure for the individual iterations that is amenable to weighted training dataset $\{x_i, y_i, w_i\}_{i=1,\ldots,n}$. Incorporating the weights w_i is straight forward when the risk is expressed as an expectation, since the emperical risk of Eq. 41.3 is just replaced with the weighted average. Similarly, the heuristic for many of the tree-based learning algorithms (*e.g.* the Gini index) also have natural generalizations with weighted events.

In the context of classification, the weighted error of the model $\hat{y}_t(x)$ is

$$\text{err}_t = \frac{\sum_i w_i^{(t)} \mathbf{1}[y_i \neq \hat{y}_t(x_i)]}{\sum_i w_i^{(t)}} \,. \tag{41.57}$$

Based on this weighted error, the coefficient β_t of the component $\hat{y}_t(x)$ in Eq. 41.56 is given by

$$\beta_t = \log\left(\frac{1-\text{err}_t}{\text{err}_t}\right) \,. \tag{41.58}$$

Then for the next iteration the weights of the misclassified events are updated as $w^{(t+1)} = w^{(t)} \exp(\beta_t)$ and then renormalized so that the sum of all weights is 1. This reweighted dataset is then used to train the next model $\hat{y}_{t+1}(x)$ and the entire procedure is initialized with uniform weights $w_i^{t=0} = 1/n$.

There is an analogous procedure for regression with the squared loss function based on the residuals $r_i = y_i - \hat{y}_t(x_i)$ (see for example Ref. [149] for details).

Gradient boosting One of the most powerful forms of tree based models, which is implemented in the tool `XGBoost` is referred to as *gradient boosting* [266]. In this setup, the model is purely additive as in the case of random forests, so the model is Eq. 41.56 with all $\beta_t = 1$. Note this is without loss of generality since the β_t can be absorbed into the b_j of Eq. 41.55. As with AdaBoost, the model is built sequentially through the sequence $\hat{y}_1, \ldots, \hat{y}_T$.

At each iteration, a new term f_t will be added to the sum in Eq. 41.56. For a given decision tree defined by splits on attributes, one can approximate the objective function (the loss function $\mathcal{L}$ plus a regularization term Ω) as a function of b_j in a second order Taylor series:

$$\text{obj}^{(t)} = \sum_{i=1}^{n} [\mathcal{L}(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + \text{constant} \,, \tag{41.59}$$

where

$$g_i = \partial_{\hat{y}_i^{(t-1)}} \mathcal{L}(y_i, \hat{y}_i^{(t-1)}) \tag{41.60}$$

and

$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} \mathcal{L}(y_i, \hat{y}_i^{(t-1)}) \tag{41.61}$$

In `XGBoost`, the regularization term is taken to be

$$\Omega(f) = \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} b_j^2 \,, \tag{41.62}$$

where J is the number of terminal nodes in the tree. With the second-order approximation of the objective, one can directly solve for the optimal b_j for the next tree and the corresponding value of the optimized objective function. The improvement in the objective function can then be used as a heuristic for choosing the best split. Specifically, define $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$, where I_j is the set of indices of data points assigned to the jth leaf. The heuristic used in `XGBoost` for splitting a node is

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right] - \gamma \,. \tag{41.63}$$

This formula can be interpreted as the score on the new left leaf plus the score on the new right leaf minus the score on the original leaf minus a regularization penalty on the additional leaf. If the gain from splitting a leaf is smaller than γ, then the total Gain is negative and the split will not be added, which can be seen as implementing a form of pruning.

41.8.4 *Neural networks*

In this section we focus on the different types of components used in modern neural network architectures. Gradient-based optimization techniques are most commonly used for training neural networks, and they are described in Sec. 41.9.1. Similarly, other important aspects to effectively training neural network models such as parameter initialization and early stopping are discussed in Sec. 41.9.

The vanishing and exploding gradient problem is a common challenge for gradient-based optimization of neural networks and is described in Sec. 41.9.5. That problem is referred to repeatedly in this section because it has motivated the development of numerous architectural components described below.

41.8.4.1 *Feed-forward multilayer perceptron*

One of the core components in neural networks is the fully-connected, feedforward network or*multilayer perceptron* (MLP), which is composed of L *layers*: $f = f^{(L)} \circ \cdots \circ f^{(1)}$. The lth layer defines a function that maps a d_{l-1}-dimensional input vector, called *features*, to an d_l-dimensional output $f^{(l)} : \mathbb{R}^{d_{l-1}} \to \mathbb{R}^{d_l}$. A unit producing an individual component of the d_l-dimensional output is called a *neuron* or a *filter* interchangeably. For $l < L$, the functions f_l are called hidden layers, and the number of neurons (d_l) is referred to as the width of the hidden layers. The layers in an MLP take on the form:

$$f^{(l)}(u) = \sigma^{(l)}(W^{(l)}u + b^{(l)}) \,, \tag{41.64}$$

where $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ is called the *weight matrix*, the components of the vector $b^{(l)} \in \mathbb{R}^{d_l}$ are referred to as the *biases*, $u \in \mathbb{R}^{d_{l-1}}$ is the input from the previous layer, $W^{(l)}u$ denotes a matrix-vector product, and $\sigma^{(l)}$ is a non-linear *activation function* that is usually applied element-wise. The parameters of the network comprise the full collection of weights and biases, $\phi = (W^{(1)}, \ldots, W^{(L)}, b^{(1)}, \ldots, b^{(L)})$.

41.8.4.2 *Activation functions*

The activation functions σ in neural networks are nonlinear functions and key to the expressiveness of the resulting family of functions. Two traditionally used functions are the *logistic or sigmoid* function $\sigma(x) = 1/(1+e^{-x})$ and *hyperbolic tangent* function $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$. These functions are bounded to be $(0, 1)$ and $(-1, 1)$ respectively, and are symmetric about the input value of zero. On the other hand, away from the zero input value, a gradient of both functions quickly vanishes and this poses a challenge in using gradient-based optimization method (see Sec. 41.9.1). This can be avoided, to some extent, by normalizing the input values and carefully initializing the values of $W^{(l)}$ and $b^{(l)}$. These are discussed in Sec. 41.9.7, 41.9.8 and 41.9.9. Yet, it becomes difficult to maintain a null input value for a *deep* neural network, a model with many layers. Instead, a popular choice for a deep neural network is the *rectified linear unit* (ReLU):

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{41.65}$$

whose computational cost is small and ensures that the gradient does not vanish for $x \in (0, +\infty)$ [267, 268]. An alternative to preserve a non-zero gradient in negative input values are called *leaky ReLU* and modifies the output to $0.01x$ for $x \in (-\infty, 0)$ [269]. Another variant, called *parametric ReLU* (PReLU), turns the coefficient 0.01 into a variable that is optimized as a part of the model during optimization [270].

The choice of activation functions depends on the model architecture and applications. As described, while the use of ReLU types are a typical choice for a deep neural network, a logistic

function is a popular choice at the final layer for classification tasks. In the area of neural scene representation, sinusoidal activation functions have been found to be surprisingly effective [271].

Recently, additional smooth loss functions have been found to work well with larger models, such as the Gaussian-error linear unit (GELU) [272],

$$\text{GELU}(x) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \tag{41.66}$$

and swish function [273]

$$\text{swish}_\beta(x) = \frac{x}{1 + e^{-\beta x}}, \tag{41.67}$$

which smoothly interpolates between a linear function ($\beta = 0$) and ReLU ($\beta = \infty$). In addition, the value of $\beta = 1$ corresponds to the sigmoid-weighted linear unit (SiLU) [274].

Softmax 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 discrete categories. Given a real-valued input vector $u \in \mathbb{R}^n$, the softmax function computes the output vector $v \in \mathbb{R}^n$ the ith component is given by:

$$v_i = \frac{\exp(u_i)}{\sum_{j=1}^{n} \exp(u_j)} \,. \tag{41.68}$$

The result has the property that $v_i \in (0, 1)$ and $\sum v_i = 1$. The components of the input vector u are often referred to as *logits* in reference to their connection to the logistic function used in logistic regression. The softmax function is commonly used as the last layer in multi-class classifier. The softmax is also used in the context of attention (see Sec. 41.8.4.6).

41.8.4.3 *Universal approximation and deep learning*

There are a number of universal approximation theorems in the theory of neural networks. One of the first was that even with one hidden layer ($L = 2$), an MLP can approximate any continuous function if the nonlinear activation function σ is not a polynomial and the width d_1 is large enough [275]. However, it is often more efficient (in terms of the number of parameters) to increase the *depth* of the network L [276].

Training a deep network (*i.e.* $L > 2$) that generalizes well can be difficult, requiring large training datasets, many gradient updates, and suitable regularization. The introduction of large labeled training sets, advances in computing (*e.g.* graphic processing units or GPUs which enabled orders of magnitude acceleration in parallel computation including matrix multiplies [277]), development of ReLU, research progress in initialization and optimization algorithms for model parameters, and regularization techniques like *dropout* [17] all played an important role in the rise of *deep learning* [2, 278]. Though the name deep learning was originally a reference to the depth L of such networks, modern deep learning is characterized more by the composition of various types of modules that are trained through gradient-based optimization. Below we introduce some other common network architectures.

41.8.4.4 *Convolutional neural networks*

Convolutional neural networks (CNNs) are widely used for image-like data. They implement the convolution of the input image u and a *filter* W (also referred to as a kernel). The parameters of the filter are learnable and the convolution involves traversing over input and calculating the inner product of the filter W with the part of the input in the *receptive field*, which has the same spatial shape as the filter and is centered at the target pixel. At each location—indexed by i and

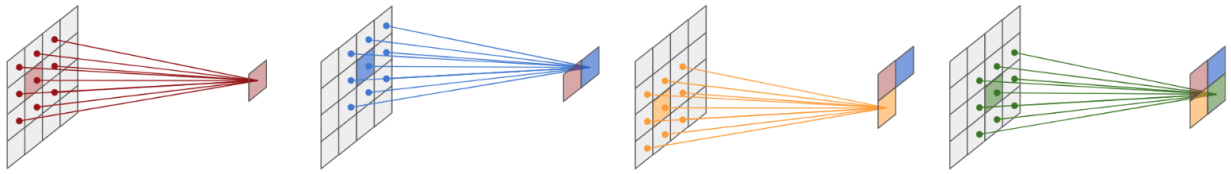

Figure 41.6: A pictorial description of a kernel convolution over four input pixels. It takes a product of the weight matrix (kernel) and the local input matrix centered at a target pixel. The operation is repeated over the input image using the same kernel. The size of the output image depends on the size of the kernel, stride, and padding. In this figure, the kernel size of 3, stride of 1, and padding of 0 is used.

j below—there is a pixel that may have a vector of features associated with it. In the context of CNNs, these components of these features—indexed by c and c' below—are often referred to as *channels* in reference to the red, green, and blue color channels in a traditional image. The convolution operation is often denoted with a $*$, and the result can be expressed as

$$v_c(j) = (W * u_c)(j) = \sum_{c'} \sum_{i} W_{c,c'}(i) u_{c'}(\text{“}j - i\text{”}) \,, \tag{41.69}$$

where “$j - i$” is shorthand for the pixel index corresponding to the translation from pixel j to i. By repeating the operation over all pixels, the result of a kernel convolution is also an image as illustrated in Fig. 41.6. Note that the the number of channels in the output v does not need to be the same as in the input, and the collection of filters $W_{c,c'}$ is often referred to as a filter bank. The entire image for a fixed channel index is often referred to as a *feature map*.

A key feature of the CNN architecture is that it is *equivariant* to translations, meaning that if the input image is shifted (*e.g.*, $u(i) \to u'(i) = u(i-k)$), then the output is also shifted by the same amount (*e.g.*, $v(j) \to v'(j) = v(j-k)$). This equivariance property is a natural consequence of using convolutions. A fully connected MLP would not generally have this symmetry; however, it is enlightening to imagine transferring the computation performed by a CNN to the weights and biases of a fully connected MLP, which would result in duplicating the weights of the filters multiple times. In this view, the CNN can be interpreted as a fully connected MLP with *shared weights*, which would maintain the equivariance property. This view is helpful for gaining intuition about the inductive bias of models and makes clear that a CNN is a subset of the fully connected MLPs that satisfy the translation equivariance property.

One may wonder how CNNs identify features with a spatial size larger than a typical kernel size. One mechanism for this is by stacking multiple convolutional layers, *e.g.*, the composition of two 3×3 kernels will lead to an effective 5×5 kernel in terms of the receptive field. In addition, a typical CNN architecture uses pooling (described below), which effectively downsamples the image so that it can be processed at different resolutions. The effective receptive field in the input image may be much larger than the kernel size in this case. An alternative approach is to use an *inception module*, which is designed to extract features simultaneously using kernels of different size [279].

Pooling *Pooling* plays an important role in convolutional neural networks both practically and in terms of their mathematical properties. A pooling operation is a type of aggregation or down-sampling that takes many pixels as input and produce one pixel for output. Typically, the pooling

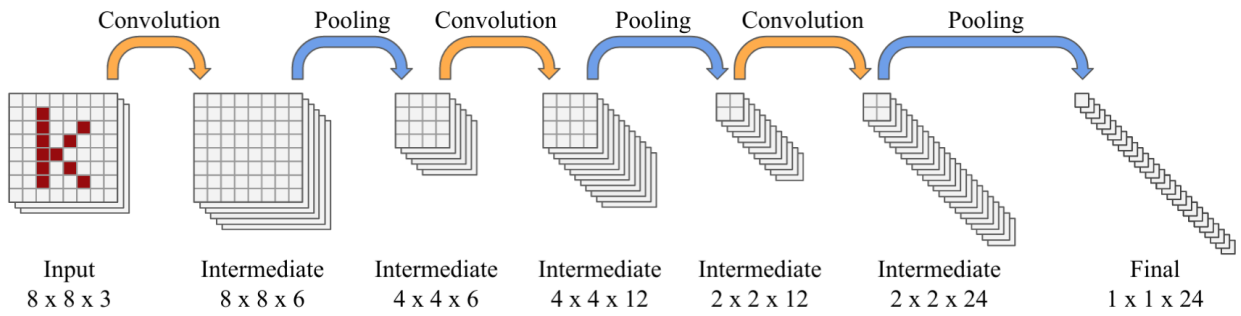

Figure 41.7: An example CNN architecture to extract a 1-dimensional array of features from an image via succession of convolution layers and pooling operations. The (square) kernel, stride, and padding size of a convolution operation are 3, 1, and 1 in respective order. The pooling operation uses a square kernel size of 2. The number of filters at the first convolution layer is 6, and is increased by a factor of 2 at subsequent convolution layers.

operation is applied independently for each channel or feature component. The most popular pooling operations are *max* and *average* pooling. Max pooling picks the highest activation pixel value within the specified receptive field, while the average pooling computes the average pixel value in the receptive field. The idea of pooling generalizes to other architectures, including graph neural networks where the receptive field includes the neighbors of a particular node in the graph (see Sec. 41.8.4.7). Pooling can make the model robust to small, local deformations in the input, a property called *geometric stability* [280–282]. This type of local deformation is important and distinct from the equivariance to rigid translations provided by the convolutional structure. Repeated pooling operations that eventually lead to a single feature vector with no spatial index is what gives rise to the invariance of common CNN architectures to translations (*i.e.*, an image with a dog will be labeled 'dog' regardless of where the dog is in the image).

CNN architectures for image analysis A typical CNN for extracting a 1-dimensional array of features is designed with repeating blocks of convolution layers and pooling operations [283]. Figure 41.7 shows an example evolution of a data tensor through the succession of convolution and pooling operations to extract a 1-dimensional array of features, which then can be fed into a block of MLP for an image classification (or a regression) task. This type of architecture is referred to as an *encoder* or *feature extractor*.

The reduction in the spatial size of an image is performed slowly, typically by a factor of 2, which is the minimum possible reduction factor. After the reduction of the spatial extent, the number of channels is typically increased (also by a factor of 2 in most cases), converging one set of feature maps into a larger number of downsampled feature maps. There may be more than one convolution layer within each spatial resolution (*i.e.*, between the grouping operations). Following these design principles, CNN encoders typically become *deep*, consisting of dozens or sometimes hundreds of convolution layers, and face challenges of vanishing gradient problem (see Sec. 41.9.5). A standard practice to mitigate this issue is to explicitly *normalize* the input tensor input in each convolution layer using algorithms such as *batch normalization*. This will be discussed in Sec. 41.9.9.

There are three main categories of computer vision tasks where CNNs are often used:

- *image classification or regression* requires a prediction of single value for the whole image (*i.e.*, a category or target value),

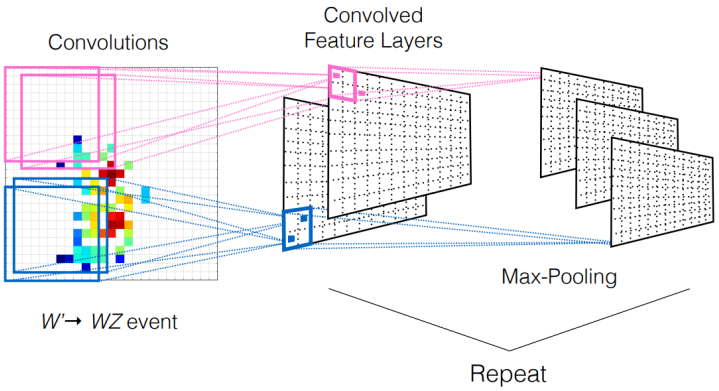

Figure 41.8: CNN classifier for identifying highly boosted W bosons at ATLAS.

- *object detection* produces a list location information, typically as a rectangular shaped bounding box, for detecting arbitrary number of objects in the input image, and
- *semantic segmentation* brings a classification task down at the pixel-level (or regression although less common) to identify the class or a feature of every pixel in an image.

As discussed previously, a CNN feature extractor followed by MLP is often used for image classification and regression tasks in wide range of applications including particle physics. Many successful CNN architectures for object detection and semantic segmentation applications share key designs which we briefly discuss below.

Region convolutional neural network (R-CNN) is one of the most successful design for object detection [284]. R-CNN has been explored in HEP experiments where the number and location of signal (*e.g.*, neutrino interactions) are not known apriori in large image data such as neutrino detectors [3, 25, 187]. R-CNN consists of multiple CNNs. The first is a feature extractor which produces a spatially compressed feature tensor. For every pixel in the compressed tensor, the second CNN applies 1×1 convolution to predict two information: an *object score* to inspect whether or not there is a target object in the (spatially compressed) pixel, and prediction of the location and size of a rectangular, axis-aligned bounding box that contains the object (if exists). This second CNN is called the *region proposal network* (RPN), and the bounding box is called the *region of interest* (ROI). For each ROI with an object score above threshold (hyperparameter), the third CNN operates in the corresponding sub-field of an already-compressed tensor (*i.e.* by the first CNN) to perform a classification for an object inside the ROI. This approach can produce multiple ROIs for the same object with a high overlap. Those predictions are reduced using non-maximum suppression (NMS) algorithm which computes the intersection-over-union (IoU) to combine overlapping ROIs that are likely detecting the same object.

Residual networks and skip connections

The expressivity of a neural network increases as more hidden layers are added, but gradient-based optimization of deep models can be notoriously difficult due to vanishing gradients (see Sec. 41.9.5). One powerful technique to address this challenge is a *residual network* (ResNet), which is a modular architecture design that can be applied to neural network models [285]. Suppose a $f(x)$ as the target transformation to be learned by a few stacked layers where x is the input to the first layer. The authors of ResNet hypothesized that it may be easier for a model to learn a residual

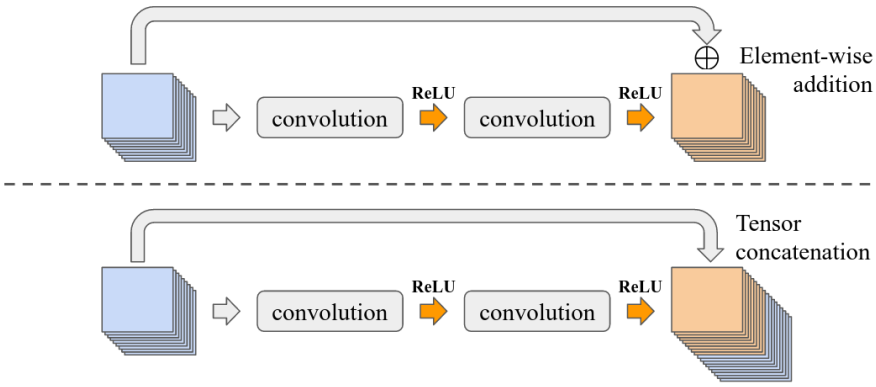

Figure 41.9: Two types of skip connections: the top is from ResNet where the input is element-wise added to the output tensor of a block of convolution layers while the bottom shows a concatenation of the input to the output tensor as employed in other models including U-Net and DenseNet.

transformation $\tilde{f}(x) \coloneqq f(x) - x$, thus the objective to learn is $\tilde{f}(x) + x$ where $\tilde{f}(x)$ denotes the output of stacked layers. This form assumes $\tilde{f}(x)$ and x share the same tensor dimension and size. If they differ in the feature dimension, equivalently the count of channels in an image tensor, one could use 1×1 convolutions to transform and match the dimension. Adding the input tensor x to the output of a convolution operation $\tilde{f}(x)$ in ResNet is a form of *skip connection*. For a residual block that outputs $y = \tilde{f}(x) + x$, the backpropagated gradient to the input x becomes

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \left(I + \frac{\partial \tilde{f}}{\partial x} \right), \tag{41.70}$$

meaning it sums an identity path with the gradient through the residual branch, helping prevent vanishing gradients. ResNet authors demonstrated performance improvement at depths exceeding 1000 layers where the non-residual counterpart could not improve beyond a few dozen layers. The ResNet design is widely used in many CNN architectures as it is modular, *i.e.*, a *residual block*, and can be applied per a stack of convolution layers (*e.g.*, U-ResNet introduced for LArTPC detectors uses ResNet modules within a U-Net architecture [26, 184]).

U-Net is one of the of most successful models used for semantic segmentation [286]. Since the output of U-Net is also an image, it is more interpretable compared to models for image-level classification or regression. The model is used widely in HEP experiments in both 2D and 3D image data [26, 27, 184, 194, 196]. The architecture of U-Net consists of a CNN encoder and *decoder*. A decoder consists of convolution and *transposed convolution* layers (also called deconvolution). The operation of a transposed convolution can be seen as the opposite of a convolution: for every input pixel, its value is multiplied by the kernel and copied to the output. In contrast to a regular convolution layer that reduces input pixels via the kernel, a transposed convolution layer broadcasts input pixels via the kernel, producing an output that is larger than the input. In the decoder of U-Net, transposed convolution layers are used to upsample spatially compressed feature tensors back to the original image resolution. Standard convolution layers are placed between upsampling operations. Features for every output pixel can be used for either a classification or a regression task. The idea behind encoder-decoder architecture is to extract features in the encoder, and the decoder interpolates those features back to the original spatial resolution. The downsampling

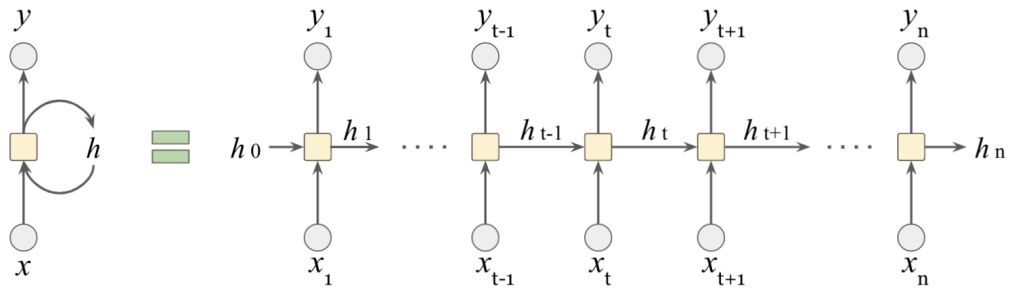

Figure 41.10: Pictorial description of a RNN (on the left) which takes an input and produces an output at every step with a hidden-to-hidden connection. The right diagram is unrolled over discrete steps. The yellow box represents a cell: a set of operations unique to each architecture.

operation (*e.g.*, max pooling) in the encoder is, however, a lossy process where spatial information is permanently lost. This is a major obstacle to achieve a high precision semantic segmentation task. The U-Net architecture overcomes this challenge by concatenating intermediate tensors in the encoder block with the tensors of the corresponding spatial size in the decoder block. This is a type of a *skip connection* discussed previously, which dramatically improves the performance of semantic segmentation.

41.8.4.5 *Recurrent neural networks*

Recurrent neural networks (RNNs) [287] are a family of neural networks designed for sequential data (*e.g.*, time series). Consider sequential data where x_t represents each step in a sequence with $t \in [1, n]$. A typical RNN takes the following form:

$$h_t = g_h(h_{t-1}, x_t, \theta) \tag{41.71}$$

where h_t and θ denote the *hidden state* of the system and parameters of g_h, the RNN model. The term *recurrent* refers the nature of the model operating on the previous state of the system (and hence the whole history). RNNs operate on three types of tasks:

- *One-to-many* takes a single input and generates a sequence (e.g. generates a sequence data, such as a sentence or waveform, given a category).
- *Many-to-one* takes a sequence and generates an output (e.g. sequence-labeling).
- *Many-to-many* takes a sequence and generates a sequence where the length of input and output sequence may be same (e.g. classification of individual element in a sequence) or different (e.g. sequence to sequence mapping).

Figure 41.10 shows an example for a many-to-many task, where $\{x_t\}_{t=1:n}$, $\{y_t\}_{t=1:n}$, and $\{h_t\}_{t=0:n}$ denote the inputs, outputs, and hidden states respectively. A set of operations at each time step is called a *cell*. A simple RNN cell may look like:

$$\begin{aligned} h_t &= g_h(Wx_t + Vh_{t-1} + b) \\ y_t &= g_o(Uh_t) \end{aligned} \tag{41.72}$$

where $W \in \mathbb{R}^{d_h \times d_i}$, $V \in \mathbb{R}^{d_h \times d_h}$, $U \in \mathbb{R}^{d_o \times d_h}$ are matrices g_h and g_o represent functions. d_i, d_h, and d_o are the dimension of input, hidden state, and output. $b \in \mathbb{R}^{d_h}$ is a bias term. An example application is sequence-labeling where the goal is for y_t to classify each input x_t in the sequence. In that case, one might use $g_h = \tanh$ and $g_o = \text{softmax}$ and use a loss function that averages classification accuracy over the sequence.

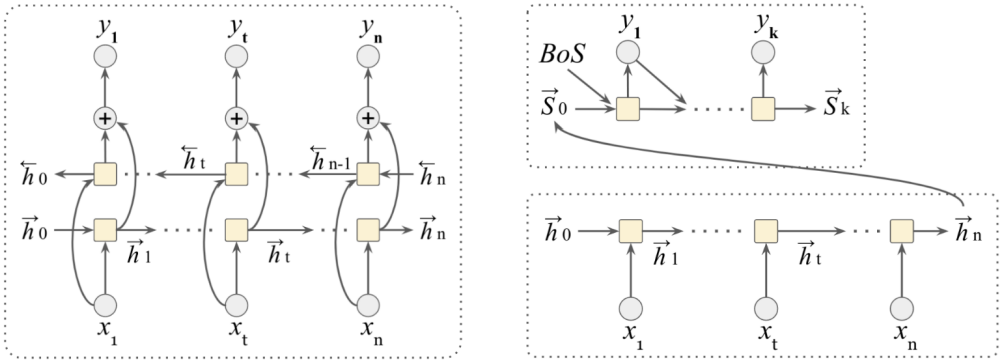

Figure 41.11: Bidirectional RNN (left) provides contexts in the preceding and subsequent parts of the input sequence. RNN encoder-decoder (right) can generate an output with a different sequence length from an input. Each cell in the decoder may take a previously generated element, starting from a special marker that signals the beginning of the sequence (BoS) and ending when the end of sequence is generated.

Variations in RNN architectures result from the design of cells (described below) and flow of information across the cells. For instance, a bidirectional RNN (Figure 41.11 left) employs two set of RNNs, one processing the sequence in the forward direction and the other in the backward direction, and the hidden states from both directions are then combined to capture the context from both parts of the sequence. An RNN encoder-decoder (Figure 41.11 right) use one RNN to generate a context vector that encodes the whole input sequence, and use a separate RNN to generate another sequence from the encoded context. This can be used for machine translation.

LSTM and GRU An RNN applies the same functions g_h and g_o in Eq. 41.72 repeatedly for each element of the sequence. This repeated component is similar to the shared weights for a convolutional filter in a CNN.

A hyperbolic tangent (tanh) is traditionally a popular choice for g_h as it regulates the magnitude of the hidden states and prevents it from diverging. Yet, this simple model is challenging to train for a long sequence of data [288, 289]. This is partially due to the fact that tanh contributes to the vanishing gradient problem and because repeated multiplication of the same weight matrices (i.e. V and W in Eq. 41.72) can lead to gradients that can either explode or vanish (see Sec. 41.9.5). Additionally, the way the signal accumulates means that changes early in the sequence have different impact from changes late in the sequence.

Long short-term memory (LSTM) [290] is a model designed to address the issue of vanishing gradient for RNNs. In this model, a *context* is introduced as a way to enable the model to hold long-term memory while the hidden states remain to hold short-term memory. The context c_t and hidden state h_t at step t are computed as follows:

$$\begin{aligned} c_t &= f_t \bigodot c_{t-1} + i_t \bigodot \tilde{c}_t \\ h_t &= o_t \bigodot c_t \end{aligned} \quad \text{where} \quad \begin{aligned} f_t &= \sigma\left(W^f x_t + V^f h_{t-1} + b^f\right) \\ i_t &= \sigma\left(W^i x_t + V^i h_{t-1} + b^i\right) \\ o_t &= \sigma\left(W^o x_t + V^o h_{t-1} + b^o\right) \\ \tilde{c}_t &= \tanh\left(W^c x_t + V^c h_{t-1} + b^c\right) \end{aligned} \tag{41.73}$$

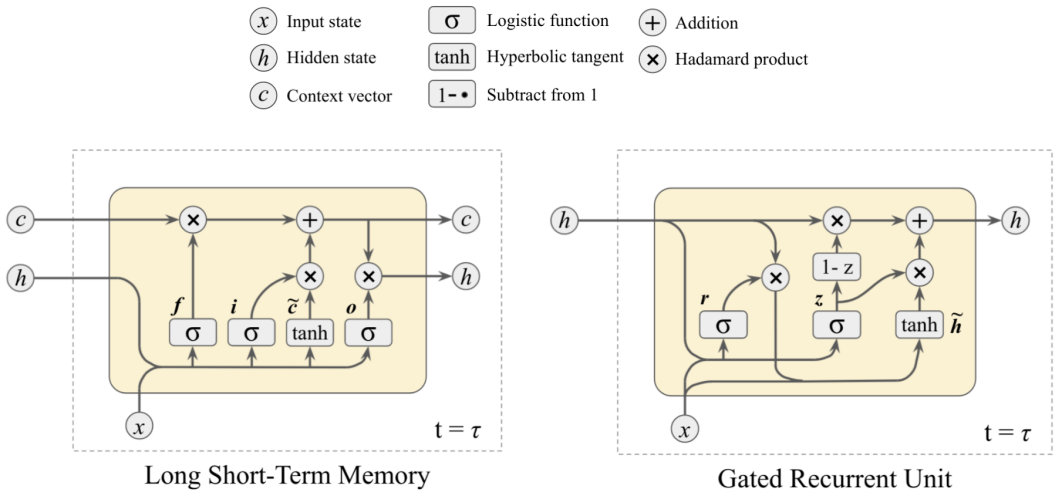

Figure 41.12: LSTM (left) and GRU (right) are both gated neural network designed to address a vanishing gradient problem for RNNs.

where σ and $\odot$ denote logistic function and an element-wise (*i.e.*, Hadamard) product and f_t, i_t, and o_t are referred to as *gates*. Each gate outputs a value between 0 and 1, and is associated with unique weights, W and V, and a bias b. One can see c_t is a combination of the previous context vector c_{t-1} and a new context vector $\tilde{c}_t$. The *forget* gate f_t controls which and how much of the past context should be kept or forgotten. The *input* gate i_t controls how much of the present context $\tilde{c}_t$ should propagate to the current state c_t. The output gate o_t controls which and how much of the context vector should represent the present hidden state h_t. From Figure 41.12, one can see that the context vector c_t evolves with a gated addition operation. As such, it can be seen as an uninterrupted path for gradients to flow. This is similar to a residual connection (see ResNet in Sec. 41.8.4.4), which enabled training of CNNs with thousands of layers.

Another gated model to solve a vanishing gradient problem is the *gated recurrent unit* (GRU) [251]. The GRU is similar to the LSTM with a few simplifications: the GRU merges the context vector and the hidden states and combines three gates into two. As a result, it requires less computational resources while retaining a similar level of performance for long sequences. The GRU operations are defined as follows:

$$h_t = z_t \odot h_{t-1} + (1 - z_t)\tilde{h}_t \qquad \text{where} \qquad \begin{aligned} r_t &= \sigma\left(W^r x_t + V^r h_{t-1} + b^r\right) \\ z_t &= \sigma\left(W^z x_t + V^z h_{t-1} + b^z\right) \\ \tilde{h}_t &= \tanh\left(W^h x_t + V^h\left(r_t \odot h_{t-1}\right) + b^h\right) \end{aligned} \tag{41.74}$$

where r_t and z_t are referred to as *reset* and *update* gate. As one can see in Figure 41.10, the reset gate in GRU performs the same task as the forget gate in LSTM by removing or reducing the elements of its memory (*i.e.* the hidden state). The update gate z_t determines the relative proportion of the previous hidden state h_{t-1} and the new context $\tilde{h}_t$ to be mixed in producing the new hidden state.

In addition to sequential data, the LSTM and GRU units can be used for data that has a tree-like structure. In this setting, the networks are often referred to as recursive neural networks or TreeRNN and they have found applications in natural language processing and jet physics [223, 224, 291–293].

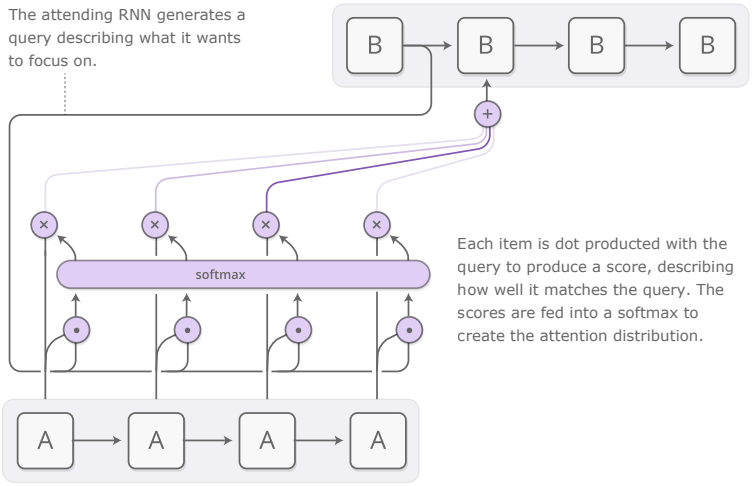

Figure 41.13: An illustration of the attention mechanism from Olah and Carter, "Attention and Augmented Recurrent Neural Networks." Lower boxes labeled `A` represent input elements in the sequence and upper boxes labeled `B` indicate output elements. The left-most line originating from the first `B` corresponds to the state s_{i-1} in the text.

41.8.4.6 *Attention and transformers*

The idea behind *attention* is to form a representation for the input, but different parts of the input are weighted differently according to the task at hand. By making the weights learnable, the network can learn to attend to the relevant parts of the input. For the ith task, one can form a task-specific context c_i by computing the weighted average of the hidden state representations h_j for each component of the input. A softmax function is used to produce attention scores α_{ij} for the jth input and ith task because it assigns a positive value to each component of the input and sums to one, $\sum_j \alpha_{ij} = 1$. Putting these ingredients together, we have the *additive attention mechanism*

$$c_i = \sum_{j=1}^{n} \alpha_{ij} h_j \qquad \text{where} \qquad \alpha_{ij} = \underset{j}{\text{softmax}}(\beta_{ij}) \;, \tag{41.75}$$

where softmax_j indicates that normalizing sum runs over the index j and the logits β_{ij} can be computed from a neural network. In the case of a cell of an RNN encoder-decoder network (see Fig. 41.11) that is decoding element i with an incoming input state s_{i-1}, the logits for the attention mechanism can be computed as

$$\beta_{ij} = U \tanh\left(W s_{i-1} + \widetilde{W} h_j + b_i\right) \;, \tag{41.76}$$

where U, W, and $\widetilde{W}$ are the weights and b is the bias term of the model. Figure 41.13 from Ref. [294] illustrates the full attention mechanism. This idea was first implemented by a model called *RNNSearch* that made a breakthrough in machine translation by combining a bidirectional RNN with an additive attention mechanism [295].

The values α_{ij} can be used to visualize the influence of the jth input element on the ith output element, which improves interpretability of the model [294] as shown in Fig. 41.14.

In additive attention (Eq. 41.75), the hidden representations h_j, also called *values*, are combined through a weighted average based on the coefficients α_{ij}, resulting in a task-specific context vector

c_i. These values are often arranged in a matrix labeled $V \in \mathbb{R}^{m \times d_v}$, where the m rows of the matrix correspond to individual hidden state vectors of length d_v. The α_{ij} can also be represented as a $n \times m$ matrix α resulting from applying the softmax function to the $n \times m$ matrix β, normalized independently for each row. With this notation, Eq. 41.75 could be rewritten as $c = \text{softmax}(\beta)V$, where the softmax is normalized per row.

One powerful and widely used variant of the attention mechanism is *scaled dot-product attention*. In scaled dot-product attention, instead of using an neural network to compute the logits β as in Eq. 41.76, the logits are computed by forming a dot product between an incoming *query* and *key*. The set of n query vectors can be arranged into the matrix $Q \in \mathbb{R}^{n \times d}$ and the set of d key vectors can be arranged into the matrix (transpose) $K^\intercal \in \mathbb{R}^{d \times m}$. One can interpret the keys as trying to detect certain types of queries and routing the attention to the relevant value. Typically, the dot-product is scaled by a factor of $1/\sqrt{d}$. The resulting task-dependent context is $c_i = \text{softmax}_j(q_i \cdot k_j/\sqrt{d})v_j$. A common, though confusing, notation is simply

$$c = \text{Attention}\,(Q, K, V) = \text{softmax}\left(\frac{QK^\intercal}{\sqrt{d}}\right)V\,, \tag{41.77}$$

where c is a $n \times d_v$ matrix organizing the n context vectors of length d_v that are tailored summaries of the input vector for each of the n tasks.

The *transformer* architecture is a powerful encoder-decoder model based on the scaled-dot product attention mechanism. It was originally designed for sequential data and subsequently used in other areas of research including computer vision. One advantage of scaled-dot product attention is that computing the attention weights does not involve any sequential processing. This allows the models to better leverage the parallelism of the hardware to train more expressive models faster than before. In place of the gated units of an RNN that are key to avoiding the vanishing gradient problem, the transformer architecture employs a residual connections at every attention module (*i.e.*, the input tensor is added to the output as in Fig. 41.9).

The second major ingredient in the transformer architecture is *multi-head attention*. A multi-head attention module executes multiple scaled dot-product attention modules in parallel. The query Q, key K, and value V matrices in each scaled dot-product attention module are obtained by applying linear transformations (with learnable weights) to the common Q, K, and V input matrices. Each of them can be considered a different (albeit related) *perspective* from which to derive attention.

For a sequence-to-sequence mapping task, the output of encoder is used to derive key K and value V matrices for the multi-head attention module in the decoder. The decoder is then responsible for mapping between the key-value features derived from the input (the encoder) and the queries from the decoder (which is still executed sequentially) in order to produce the final decoded output.

Finally, we note that the transformer architecture does not just employ an attention mechanism in the decoder. By employing attention in the encoder as well the model has more capacity to "interpret" the input—a concept referred to as *self-attention*. Transformer models have contributed to breakthroughs in many areas of scientific and industrial research [296]. While transformers are very powerful, they also require a larger number of training samples due to the weaker inductive bias than other models.

41.8.4.7 *Graph networks and geometric deep learning*

Graphs are a powerful archetype for representing structure data. A graph consists of *nodes* as elements and *edges* between between them. Graphs are sufficiently flexible to describe many types of structured data including images and sequences. Graph-based neural networks can also

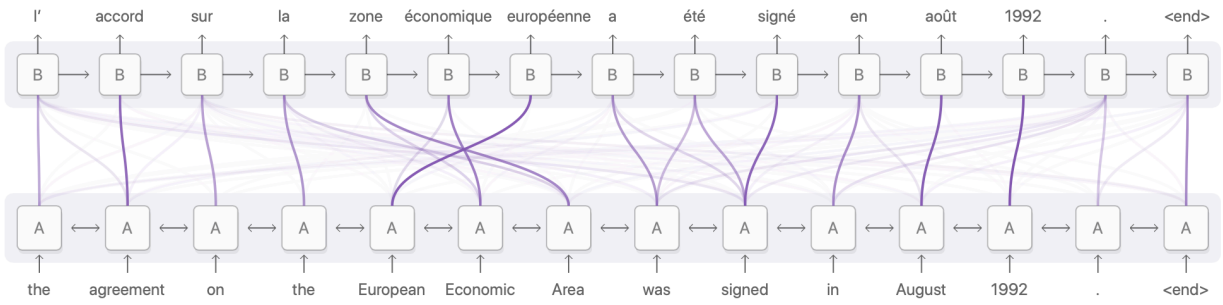

Figure 41.14: Visualization of the attention weights in a sequence-to-sequence problem from Olah and Carter, "Attention and Augmented Recurrent Neural Networks." The thickness of the lines is proportional to the attention weights α_{ij}.

be seen as a generalization of many common types of machine learning models such as recurrent and convolutional neural networks [282]. The term *geometric deep learning* refers to this recent formulation that focuses largely on the symmetries of the data.

An earlier attempt to organize the variations on different flavors of graph-based neural networks can be found in Ref. [297]. In their formalism, a graph network may be represented as $G(\mathbf{u}, V, E)$ where $\mathbf{u}$ represents an array of global features, $V = \{\mathbf{v}_i\}_{i=1:N^v}$ represents a set of N^v nodes with $\mathbf{v}_i$ as features for the ith node (e.g. such as RGB channels if a node represents a pixel in image data), and $E = \{(\mathbf{e}_k, r_k, s_k)\}_{k=1:N^e}$ represents a set of N^e edges with $\mathbf{e}_k$ as features for the kth edge. An edge may be (bi)directional where r_k and s_k denotes the destination and origin nodes respectively. The features of a graph may evolve with three *update* functions ϕ and three *aggregate* functions ρ:

$$\begin{array}{lll} \mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}) & \mathbf{e}'_i = \rho^{e\to v}(E'_i) & \text{where} \quad E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e} \\ \mathbf{v}'_i = \phi^v(\bar{\mathbf{e}}_i, \mathbf{v}_i, \mathbf{u}) & \bar{\mathbf{e}}' = \rho^{e\to u}(E') & \text{where} \quad E' = \cup_i E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e} \\ \mathbf{u}' = \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}) & \bar{\mathbf{v}}' = \rho^{v\to u}(V') & \text{where} \quad V' = \{\mathbf{v}'_i\}_{i=1:N^v} \end{array} \tag{41.78}$$

where $\mathbf{e}'$, $\mathbf{v}'$, and $\mathbf{u}'$ denote the updated node, edge, and graph features. In Graph Networks, three types of information are updated in the following order. The first step is ϕ^e to update every edge. The second step updates every node: for ith node, compute $\rho^{e\to v}$ to aggregate updated attributes from the edges with $r_k = i$ then compute ϕ^v to update the node attributes. The third step updates the graph attributes through ϕ^u which takes the original state $\mathbf{u}$, aggregated node and edge attributes by $\rho^{v\to u}$ and $\rho^{e\to u}$ respectively.

Graph neural networks [298] (GNNs) are the class of neural networks that work on graph-structured data. A related data format in computer vision and physics is the *point cloud*, which is an unordered set of points (*i.e.*, a graph with no edges). Operations on point cloud need to be permutation invariant (*e.g.* min, max, $+$, $\cdot$), and analysis of 3-dimensional physical object represented by point cloud need to be rotation and translation invariant as in the case for an image. PointNet [299, 300], a GNN that performs an object classification on point cloud of 3-dimensional positions, treats each point as a node, applies MLPs as ϕ^v to update node features, and global max-pooling operation as $\rho^{v\to u}$. There is no explicit edge definition in PointNet (though the model applies affine transformation to all points using spatial transformer network [301], which could be considered as a separate graph operation, to introduce rotation and translation invariance and to capture topological features). Deep sets [302] follow the same manner except ϕ^v takes the global entities $\mathbf{u}$. This is same for PointNet when performing point cloud segmentation: ϕ^v

takes a step of simply concatenating $\mathbf{u}$ to node entities to combine a local and global features. Dynamic graph CNN [303] is a variant that (re)define edges dynamically using attention mechanism: $\rho^{e\to v}$ aggregates k neighbor nodes where the inter-node distance is defined as a Cartesian distance in the feature space. ϕ^v remains a MLP and, while edges are defined, there is no associated entity. A similar technique is used in nonlocal neural network [304] to efficiently propagate local feature information to points that may be far in the 3D cartesian coordinate. Message-passing neural network [305] (MPNN) explicitly defines a feature vector as edge entities. In MPNN, $\rho^{e\to v}$ performs element-wise sum of features and feed into ϕ^e, explicitly passing features across nodes as the name suggests. While these are representative models that are frequently used in particle physics applications [30, 190, 201, 211, 212, 226, 252, 306–308, 308], it is only a tiny fraction of GNN models developed over the past decade.

Graph-based models are particularly interesting for science applications because they offer a natural way to organize the entities in the data and encode how those components interact each other. This particular type of inductive bias is referred to as *relational inductive bias* in Ref. [298]. Graph edges may be intrinsically defined in the data (*e.g.*, when representing a social network) or not (*e.g.*, a point cloud). In the latter case, the graph structure must be chosen. A naive approach may be defining a fully-connected graph. However, for applications on hundreds of thousands of nodes (*e.g.*, high resolution 3D point cloud), this may require a prohibitive amount of memory and computation. On the other hand, if the graph is too sparse, it may negatively impact the performance. One may need to compare the model performance among differently constructed graphs and balance against computational burden. Ideally, the graph would be based on some knowledge of the interactions, but in the absence of such knowledge, popular graph construction methods include fully-connected, k-nearest neighbors, a Delaunay graph, minimum spanning tree, and locality-sensitive hashing [309].

Classification and regression tasks for graphs can be formulated such that the prediction is made for the entire graph or its individual nodes or edges. Graph-level prediction is like classifying an entire image, while node-level prediction is like semantic segmentation where individual pixels are classified. For clustering of points, GNNs can approximate a transformation function for nodes into the latent space where an optimal clustering of points can be performed. For instance, Ref. [310] proposes the *object condensation* approach to extract particle information from a graph of detector measurements as well as grouping of the measurements. The model predicts the properties of a smaller number of particles than there are measurements, in essence reducing the graph without explicit assumptions on the number of targeted particles. Certain nodes are chosen to be the "condensation" point of a particle, to which the target properties are attached. A special loss function mimics attractive and repulsive electromagnetic potentials to ensure nodes belonging to the same particle are close in the latent space. Alternatively, one can formulate clustering as an edge classification task [29–32]. See Ref. [311] for a comprehensive review on particle physics applications.

41.8.5 *Model design with physics inductive bias*

Designing neural networks that respect the structure of particle physics can materially improve sample efficiency, robustness to systematics, and physical interpretability. Rather than relying on generic inductive biases (*e.g.*, translation equivariance in image CNNs), particle-physics–informed models hard-wire symmetries, conservation laws, and kinematics into the architecture or loss. This reduces the hypothesis space to functions that are a priori plausible, which is particularly valuable when training data is scarce, for extrapolation outside the training distribution, and for tasks where trust and uncertainty quantification matter as much as raw accuracy.

Exploitation of symmetry groups and hence *equivariance* is arguably one of the most important

forms of physics-informed approaches. At the constituent level, events and jets are sets of particles, so permutation symmetry is fundamental. Deep sets and graph neural networks implement this by aggregating over particles with symmetric operations [211]. Beyond permutation symmetry, Lorentz symmetry is the natural arena for high energy physics experiments. Networks can be built from Lorentz scalars and tensors or by representing features as four-vectors and ensuring intermediate outputs transform equivariantly under boosts and rotations [312–318]. Gauge symmetry offers another avenue where symmetry-aware design pays off. In lattice gauge theory, gauge-equivariant networks ensures locality and exact invariance under gauge transformations [209, 319–322].

While these models explicitly integrate laws of physics into mathematical operations and model architecture designs, it is also possible to implicitly enforce physics constraints through a loss definition and model optimization method. For instance, physics-informed neural networks (PINNs) introduce regularization terms that force predicted physics quantities to follow laws of physics in the form of partial differentiable equations (*e.g.*, acceleration as the time derivative of velocity) [323, 324]. Variants of PINNs incorporate differentiable physics models as a part of a model architectures [325]. Finally, it is also possible to introduce physics constraints in an optimization process. For example, for a machine learning model for data reconstruction, an output of the model may go through a forward physics simulator that infers the original input to the reconstruction model. By minimizing the difference between the original input and the inferred one, the reconstruction model is forced to learn a solution that is consistent with the forward physics model [326].

41.9 Learning algorithms

41.9.1 *Gradient-based optimization*

Given a parameterized model $f(x, \theta)$ and a loss function $\mathcal{L}(x, \theta)$, where x and θ denotes data and model parameters, one way to optimize θ is to first apply an appropriate initialization, $\theta_{t=0}$ (e.g. Sec. 41.9.7 for neural networks), and perform an iterative update:

$$\theta_t = \theta_{t-1} - \lambda \nabla_\theta \mathcal{L}(x, \theta), \tag{41.79}$$

where λ is a small, real valued hyperparameter called *learning rate*. To see how this works, define $\delta\theta \equiv \theta_t - \theta_{t-1}$ and consider $\delta(\nabla_\theta \mathcal{L}(x, \theta))$:

$$\delta\left(\nabla_\theta \mathcal{L}(x, \theta)\right) \approx \delta\theta \cdot \nabla_\theta \mathcal{L}(x, \theta) = -\lambda |\nabla_\theta \mathcal{L}(x, \theta)|^2 \tag{41.80}$$

which would monotonically decrease the loss function, and locally move the parameter values in the desired direction of loss function minimization. This algorithm is called *gradient descent* (GD). We note that λ needs to be sufficiently small for the approximation to hold. When λ is too large, this can be a cause of a gradient explosion discussed in Sec. 41.9.7.

41.9.2 *Stochastic gradient descent*

Stochastic gradient descent (SGD) follows GD but replaces the exact gradient term $\nabla_\theta \mathcal{L}(x, \theta)$ with a stochastic approximation, where we subsample the data in the loss function using N samples, where $N < n$,

$$\nabla_\theta \mathbb{E}_{\hat{p}(x)} \mathcal{L} \approx \frac{1}{N} \sum_i^N \nabla_\theta \mathcal{L}_i, \tag{41.81}$$

where $\mathcal{L}_i$ is the loss function for data sample i. It should be noted that N needs to be randomly and independently sampled for the approximation to hold. Implementation of SGD follows three steps: take new samples of size N, approximate the gradient, then update the parameters θ.

In the case of optimizing the loss using a static database (*i.e.* one cannot take new N samples for every update), *mini-batch learning* is often employed. This replaces the first step with a randomly

sampled *batch* of data, which is a subset of all the samples in the database. In this case, however, since a batch of data used for each parameter update is not entirely independent, a model may overfit. In practice, a part of the whole dataset is reserved as a *validation* sample, and the model performance is carefully monitored during the optimization process to avoid overfitting via an early stopping criterion (see Sec. 41.9.6 and Fig. 41.15).

SGD with slowly decreasing learning rate can be shown to converge to a local minimum almost surely under mild conditions, and to a global minimum for unimodal loss functions. SGD may also prevent getting stuck in shallow local minima of the loss function, thereby reaching a better local minimum for multi-modal loss functions. The noise in SGD with a constant learning rate can be viewed as a form of Langevin dynamics, which under proper conditions on the learning rate and mini-batch size converges to the stationary posterior distribution of the weights [327]. Thus SGD at a constant learning rate can be viewed as a sampler bouncing around and exploring the posterior surface for better solutions, descending onto the best found solution as the learning rate is decreased, a process related to temperature annealing in global optimization.

Another advantage of SGD is simply the computational cost: rather than evaluating the loss over all the data samples at each update, we use a small subset of data instead at each update. Furthermore, mini-batching can take advantage of vectorization libraries and GPU architectures. Large batch training requires specialized methods of training, such as layer-wise adaptive rate scaling (LARS) [328].

41.9.3 *Optimization algorithms*

GD and SGD are the basic building blocks for more advanced optimization algorithms. One can improve the convergence rate of gradient based optimization by considering the learning rate λ to depend on individual θ_i. Second order algorithms such as Newton's method take into account second order derivatives (Hessian) to find the minimum, and give an exact solution in a single update when the loss is quadratic around the peak. However, this requires a matrix inversion of the Hessian, which is exceedingly expensive in ML applications, where the number of network parameters is very large. As a consequence, second order optimization is rarely used in ML.

There are several improvements to the basic SGD even in the absence of Hessian information. Momentum based optimization takes a physics perspective of a viscous fluid in an external potential, where one updates current velocity with the potential gradient (force), followed by an update in position based on velocity. This approach therefore uses previous gradients in addition to the current one to compute a running average of the gradient, with a forgetting factor that controls how far back the averaging goes. This helps move faster towards the minimum in ravines, where gradient descent is usually inefficient due to the high condition number of the Hessian.

Beyond SGD with momentum, modern optimizers adapt step sizes across parameters or layers using recent gradient statistics and decouple regularization from the update. RMSprop tracks a running RMS of gradients and scales steps inversely to damp oscillations. Adam adds momentum (first moment) and variance (second moment) estimates to provide per-parameter adaptive updates. AdamW [329] improves Adam [330] by decoupling weight decay from the adaptive step, applying true L2 regularization directly to weights, which typically yields better generalization and has become a common default in vision and language models. For very large-batch regimes, layer-wise trust-ratio methods such as LARS (Layer-wise Adaptive Rate Scaling) [328] and LAMB (Layer-wise Adaptive Moments optimizer for Batch training) [331] scale each layer's effective learning rate by the ratio of parameter norm to gradient norm, enabling stable training with batch sizes of tens of thousands. More recently, Lion [332] uses the sign of the momentum update instead of a second-moment estimate, reducing memory and often improving speed and generalization; it can also be combined with decoupled weight decay. In practice, these optimizers are paired with warmup,

cosine or exponential decay schedules, and sometimes gradient clipping; the best choice depends on model size, data regime, and whether you prioritize fast convergence, large-batch scalability, or final generalization.

41.9.4 *Automatic differentiation and backpropagation*

In practice, $f(x, \theta)$ might take a complex form and may include a large set of parameters. The term $\nabla_\theta \mathcal{L} = \nabla_\theta \mathcal{L}(f(x, \theta))$ requires computing partial derivatives with respect to individual parameter θ_i. If f is a composite model (i.e. $f = f_n(f_{n-1}(\cdots, \theta_{n-1}), \theta_n)$), and if all of $f_{i:1,n}$ are differentiable, a chain rule can be applied:

$$\nabla_{\theta_i} \mathcal{L} = \frac{\partial \mathcal{L}(f(x, \theta))}{\partial \theta_i} = \frac{\partial \mathcal{L}}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdots \frac{\partial x_i}{\partial \theta_i} \tag{41.82}$$

where x_n denotes the output of nth composite function f_n. In order to compute $\nabla_{\theta_i} \mathcal{L}$ for f_i, it needs computation of a gradient at all preceding (or subsequent if seen in the forward context) functions. As the gradients accumulate across differentiable functions in the reverse order of the model composition, this technique is called *backpropagation* [287]. An example of f that satisfies conditions to apply backpropagation is a neural network, which consists of repeating blocks of a (differentiable) activation function and an affine transformation.

When the model $f(x, \theta)$ is implemented as a computer program in practice, *automatic differentiation* (AD), also called *algorithmic differentiation*, is used to compute the derivatives. AD exploits the fact that any computer program consists of a sequence of elementary arithmetic operations (*i.e.*, addition, subtraction, multiplication, and division) and functions (*e.g.*, log, exp, sin, and cos) and apply chain rules to compute the target derivative. AD has advantages over traditional approaches including symbolic and numerical differentiation. The symbolic differentiation faces a serious difficulty of converting a program into a single expression, and the numerical differentiation suffers from round-off errors. Finally, both methods scale poorly in speed of computation for calculating partial derivatives with a large number of inputs. AD delivers much faster speed and does not suffer from increasing errors for calculating higher derivatives.

There are two modes of AD: the *forward* and *backward* mode. Consider a composite function $f(x, \theta) = f_n(f_{n-1}(\cdots f_1(x, \theta_1) \cdots), \theta_{n-1}), \theta_n)$. The forward mode applies the chain rule in the same order of the forward evaluation of f by computing $\partial f_1 / \partial x$ first, then $\partial f_2 / \partial f_1$, and continue to $\partial f_n / \partial f_{n-1}$. The backward mode traverses the reverse direction: starting from the last (outer-most) function $\partial f_n / \partial f_{n-1}$, next $\partial f_{n-1} / \partial f_{n-2}$, and continue to $\partial f_1 / \partial x$. Therefore, the backpropagation of gradients can be implemented using the backward AD, in which the target variable to be differentiated is fixed and the derivative is computed with respect to each sub-expression recursively as shown in Eq. 41.82. The forward mode is simpler to implement as the order of gradient calculation follows the order of composite functions to be executed. The reverse mode typically requires less amount of computation than the forward mode, but more memory is required to store intermediate function output values to calculate derivatives efficiently. Another consideration is the mapping of dimensionality $f : \mathbb{R}^k \to \mathbb{R}^\ell$ as it concerns the number of variables to sweep from each end. The forward mode is efficient when $k \ll \ell$ while the reverse mode takes an advantage if $\ell \ll k$. For instance, in the case of an image classification where $(k, \ell) = (\text{pixel count}, 1)$, the reverse AD is more efficient.

Development of a differentiable physics simulator is an active area of research and AD-enabled programming frameworks are at the core of those research work. AD-enabled simulator can be used to solve an inverse problem of inferring the physics model parameters (e.g. calibration) or the input (i.e. reconstruction) [333, 334]. A fully differentiable physics simulator often requires, however, a custom algorithm to approximate gradients to handle cases where gradient calculation

is not straightforward (e.g. due to stochastic processes) [335]. Beyond AD, a specifically designed neural network that ensures accurate gradient calculation is also frequently used, sometimes in combination with AD-enabled framework [326, 336].

41.9.5 *The vanishing and exploding gradient problems*

Gradient based optimization crucially depends on the size of gradient with respect to each model parameter. If the magnitude of gradient is too large with respect to the distance to an optimal parameter value, it may repeatedly overshoot the target and cause an oscillation preventing convergence. If the gradient is too small, it may take an impractically long time to converge. As shown in Eq. 41.82, the gradient of ith function f_i is a product of gradients from the subsequent functions. If those gradients are too large or too small, the magnitude can can either increase or decrease exponentially in the number of layers. These are called *exploding* and *vanishing* gradient problem respectively.

Modern deep neural networks consist with many composite functions (*i.e.*, layers) and are particularly prone to this effect. Let us consider a simple RNN. From Eq. 41.72, we can write the backpropagating gradient:

$$\frac{\partial h_t}{\partial h_{t-1}} = \text{diagonal}\left(f'\left(Wx_t + Vh_{t-1} + b1\right)\right) W \tag{41.83}$$

where f' denotes the derivative of an activation function. The gradient of the contribution to the loss $\mathcal{L}_i$ from the ith element in the sequence with respect to the jth hidden state h_j is therefore:

$$\frac{\partial \mathcal{L}_i}{\partial h_j} = \frac{\partial \mathcal{L}_i}{\partial h_i} V^{i-j} \prod_{j<t\leq i} \text{diagonal}\left(f'\left(Wx_t + Vh_{t-1} + b1\right)\right) \tag{41.84}$$

where we can see that V contributes multiplicatively with $i-j$ powers when $i-j>1$. This example is explored in depth for recurrent models [288, 289] but is common for all types of deep neural networks.

In practice, one may explicitly inspect the magnitude of gradients propagating across layers to ensure an effective optimization. One way to mitigate an exploding gradient is to set the maximum gradient value $\delta_{\max}$ as a model hyperparameter and *clip* any larger gradients δ where it appears in the backpropagation:

$$\delta = \frac{\delta_{\max}}{\|\delta\|}\delta \text{if} \|\delta\| > \delta_{\max}. \tag{41.85}$$

This is called *gradient clipping* [289].

Alternatively, there are many architecture designs that are motivated by the vanishing and exploding gradient problem or which aim to help propagate gradients across many layers. These considerations drove the design of gated models like the LSTM and GRU for sequential data and also motivated the ReLU non-linearity. Other example architectural designs or components motivated by these considerations include identity mapping and skip connections used in ResNet, U-Net, and DenseNet, which allow gradients to flow across many layers.

Other factors contributing to vanishing and exploding gradient include initialization of model parameters and normalization of input data. These factors contribute in keeping the magnitude of activation, which also concerns the magnitude of gradient, within a reasonable range. A recommended practice for a gradient-based optimization of a neural network is to maintain the input values centered around zero and a similar level of covariance across the inputs (and the outputs that are the inputs to the next layer) [337]. These factors are discussed in the following.

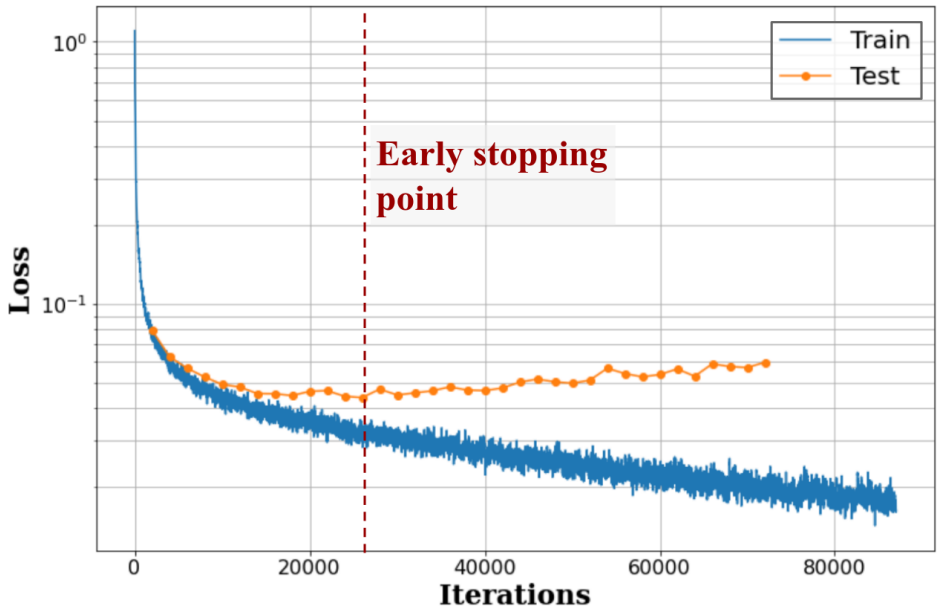

Figure 41.15: An example instance of overfitting. The training loss (vertical axis) shown in blue decreasing over iterations (horizontal axis) while the loss values evaluated on test samples shown in orange start to increase at around 26,000 iterations as indicated by the vertical line.

41.9.6 *Early stopping*

Early stopping is a form of regularization used to avoid overfitting when an iterative method, such as gradient descent, is used as a learning algorithm. Imagine a plot of the training loss and test loss as a function of iterations (*i.e.* parameter updates). As learning proceeds, the training loss will generally decrease. However, the test loss will often decrease initially and then start to increase, which is the classic sign of overfitting as shown in Fig. 41.15. The basic idea of early stopping is simply to stop training before overfitting takes place. In some approaches to early stopping theoretical analysis of the learning problem provides a prescription for when to stop the training [338]; however, the most straight forward approaches use a held-out validation dataset to monitor the generalization performance [339].

41.9.7 *Initialization of model parameters*

An improper initialization can slow down the optimization process or even result in a loss of convergence. While $b^{(l)}$ is typically initialized to zero, $W^{(l)}$ values need to be stochastic to avoid identical updates during optimization. One way is to sample $W^{(l)}$ from a zero-centered Gaussian distribution with a small variance (e.g. 0.01) [340]. However, this method does not guarantee the same variance in the input to each layer, which depends on the size of the input layer, and makes it difficult to train a deep neural network [283]. The *Glorot* or *Xavier* initialization takes this into account and sets the variance of a Gaussian distribution to be $\sigma^2 = 1/d^{(l-1)}$ assuming a symmetric activation function around zero, such as a logistic function or hyperbolic tangent [341]. The *He* initialization uses the variance $\sigma^2 = 0.5/d^{(l-1)}$, and is a simple extension of Xavier initialization for leaky, parametric, and standard ReLU activation [270].

41.9.8 *Input normalization*

Input data to a neural network is often pre-processed for the same goals discussed previously: values are shifted to have the mean of zero and scaled to keep a similar covariance across features.

Furthermore, a data may be transformed using techniques including PCA and whitening (sphering) to keep input features independent and uncorrelated from each other [337].

41.9.9 *Batch normalization*

Even with careful normalization of the input data and initialization of model parameters, the mean and covariance of the data representations in hidden layers will evolve during training and may pose challenges for learning for downstream layers. This is called an *internal covariate shift* [342] and may cause negative effects to an optimization process. Accordingly, techniques to explicitly normalize features in between hidden layers are often employed for a deep neural network. One of them is *batch normalization (BN)*, which shifts and scales the input to a hidden layer:

$$\tilde{u}^{(l)} = \gamma \frac{u^{(l)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \tag{41.86}$$

where $u^{(l)}$ and $\tilde{u}^{(l)}$ refer to the raw and normalized input to the lth layer, μ_B and σ_B represent the mean and mean-squared-error of $u^{(l)}$ calculated using a *batch* of input data used to update the network parameters. γ and β are part of model parameters that are updated during the optimization. After optimization is complete, these parameters are fixed for model evaluation during production. ϵ is a small, fixed constant value to ensure numerical stability. While it is popular (especially in computer vision), a downside of BN is its dependency on the batch size. In situations where the batch size is limited to be a small number (*e.g.*, memory limitation for a large data or a model), the performance using BN could degrade since β and γ values may not be generalized for the dataset during training.

There are several variants to batch normalization with considerations on how to group a subset of values in u^l. For instance, an image naturally has three groupings: a set of pixels across spatial axis, features within one pixel (*i.e.*, image *channels*), alongside with a grouping across multiple images (*i.e.*, batch). Different groupings have been studied and found and some are found effective to particular type of applications: layer normalization groups values along the channel and spatial dimensions [343], instance normalization groups along the spatial dimension but not along the batch nor channel [344], and group normalization is similar to layer normalization but forms multiple subgroups of channels [345]. These variants do not apply normalization across samples within a batch, and thus are agnostic to the batch size.

41.9.10 *Transfer learning: pre-training and fine-tuning*

Transfer learning is a technique to improve performance and accelerate optimization process by reusing a pre-trained machine learning model for a new task. The two tasks and corresponding datasets may differ, but fundamental features, such as implicitly learned symmetries in the underlying data, may be reusable across tasks and datasets. Transfer learning typically takes two steps: the first is to alter the model or data if necessary, then continue updating some or all of the model parameters on the new data or task, *fine-tuning* the model. The first step is required, for example, when solving a different task that requires a different architecture (*e.g.*, regression vs. classification), or when input data format requires a change (*e.g.*, original model trained on three channel image, such as RGB images, while new data has a single channel). Transfer learning has been widely practiced in the field of computer vision where large, labeled data sets are available [346–349]: a CNN trained for classifying images of an animal can be largely reused for object detection, or even for analyzing image data in science (*e.g.*, particle trajectories recorded by an imaging detector). It is a critical aspect for the development of general AI as well as interdisciplinary sharing of models across research fields.

While transformers were initially introduced for machine translation, later models such as

GPT [86,350] and BERT [85] showed that these models can be generalized to multiple NLP tasks through transfer learning by *pre-training* for several seemingly different tasks, including sentence classification, semantic similarity, question answering, and commonsense reasoning [350]. These models are collectively referred to as large language models (LLMs) and some have been fine-tuned for physics domains [351,352].

41.9.11 *Foundation models*

A key to successful transfer learning is an effective pre-training process through which general (and thus re-usable) representations are learned by a model. When a large, comprehensive dataset within a certain domain is combined with an effective representation learning method, typically using self-supervision or multi-task supervision, which forces a model to learn foundational concepts, the resulting model may potentially be generalized for any task defined in the data domain. Such AI models are referred to as *foundation models* (FMs), which are a central theme of general AI research today [353]. The LLMs such as GPTs are the first and the most successful FMs to date. The core of LLM pre-training is based on self-supervised learning (see Sec. 41.4), where, *e.g.*, a model is tasked to predict the next or missing word in a sentence. To solve this task, a model must learn not only grammar or parts of speech but also a concept of visual colors and the probability distribution over possible colors an apple can take. Using the vast online corpus as a training data, LLMs are trained to learn foundational representations of the world interpreted and generated by humans in the form of texts.

The critical properties of FMs include the *emergence*, *homogenization*, and *scalability* [353]. Emergence means the system behavior is implicitly induced rather than explicitly constructed (*e.g.*, a model's capability to generalize through self-supervised pre-training). Homogenization implies a single model or a system that can perform multiple tasks. Scalability is improvement in model performance when increasing the computing resources, number of model parameters, and size of training dataset. These properties are also used to evaluate the quality of FMs. Following the initial success of LLMs, R&D of FMs has expanded to computer vision and audio data domains.

Many modern FMs combine multiple data modalities (*e.g.*, image generation from text input, audio generation from text and image input) [98,354,355]. Multi-modal FMs are trained to cor-

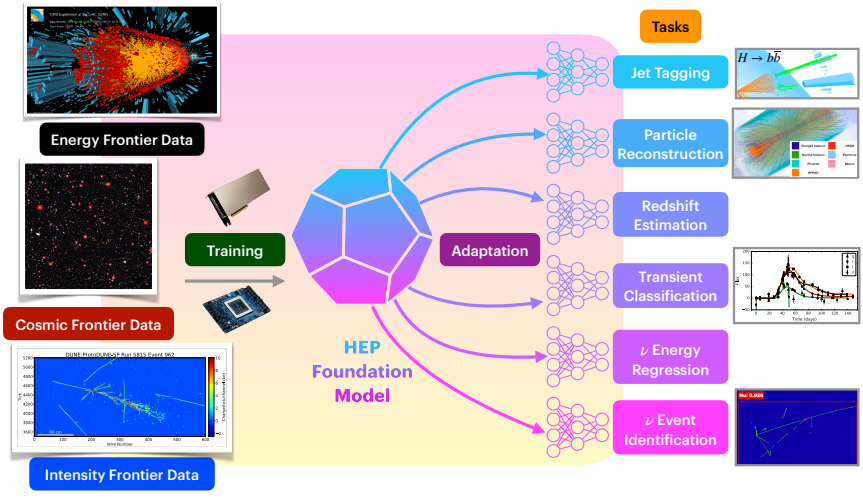

Figure 41.16: Schematic of a foundation model based on HEP data. Data is used to pre-train a large model, which can then be fine-tuned (or adapted) to various downstream tasks.

relate features from different data modalities either during a pre-training or fine-tuning stage. For example, the CLIP model achieves this by minimizing the distance between extracted features from an image and its corresponding caption (text) data [98]. It should be noted, however, pre-training FMs on sensory data (*e.g.*, 1D waveforms, 2D images, or 3D scenes) is more difficult compared to symbolic data (*e.g.*, language, math, or high-level physics data) as discussed in Sec. 41.4.

HEP datasets present unique R&D opportunities to advance understanding of FMs. Particles in high energy collisions follow well-established physics models and can be learned by FMs similar to words in natural language [87, 88]. Large public datasets of galaxies recorded in multiple modalities (*e.g.*, images and spectroscopy data) enable a contrastive learning approach based on CLIP [99]. Similarly, a high-fidelity simulator in HEP can be used to formulate contrastive learning objectives across different scenarios in a stochastic process [106]. Finally, HEP detectors offer big, high-precision data sets with challenging tasks to extract complicated physics information [96]. A schematic of a foundation model based on HEP data is shown in Fig. 41.16.

41.10 Incorporating uncertainty

A fundamental aspect of data analysis is the quantification of uncertainty. This broad topic includes the traditional distinction between statistical and systematic uncertainty, procedures for propagation of errors, and the incorporation of uncertainty in to the statistical models (*e.g.* with nuisance parameters) that are used in Bayesian or frequentist statistical procedures (see Sec. 40). Accounting for systematic uncertainty can be seen as a requirement, but ideally systematic uncertainties are also taken into account in the design of the analysis so as to mitigate their effect. The introduction of machine learning into the analysis pipeline requires revisiting the techniques used for uncertainty quantification and exposes many fundamental issues that have nothing to do with the use of machine learning per se. See Ref. [356] for a recent review on this topic.

In machine learning research and industrial settings, the mismatch between the data distribution $p_{\text{train}}(x, y)$ used for training and the data distribution $p_{\text{prod}}(x, y)$ that the model will be applied to in production is referred to as *covariate shift* or *domain shift*. For example, one might train a classifier to identify cats and dogs with images from a well lit studio and then apply the classifier on images taken in doors with poor lighting conditions and a scratched lens. Not surprisingly, the mis-classification rate of the classifier will be different between the two settings.

Physicists are keenly aware that the simulations that we use to describe the data are not perfect, and this mismodeling corresponds to a large fraction of the of systematic uncertainties accounted for in published works. Since simulated data is often used to train machine learning models (*i.e.* $p_{\text{train}}(x, y)$), it is important to understand and account for how this mismodeling will influence results when applied to real data (*i.e.* $p_{\text{prod}}(x, y)$).

One of the primary approaches to incorporating this type of uncertainty is to introduce nuisance parameters ν corresponding to the uncertain inputs to the simulation. One then parametrizes various types of perturbations (*e.g.*, corrections to efficiencies or energy scales) in the hopes that the resulting family of distributions $p(x|y, \nu)$ is flexible enough to encompass the true data distribution for class y. In this approach one does not have just two "domains" for the data (*i.e.*, p_{train} and p_{prod}), but a continuous family of domains parameterized by the nuisance parameters ν.

With this framing in mind, there are several approaches to incorporating uncertainty into an analysis that includes ML-based components:

- **propagation of errors:** one works with a model $f(x)$ and simply characterizes how uncertainty in the data distribution propagate through the function to the down-stream task irrespective of how it was trained.
- **domain adaptation:** one incorporates knowledge of the distribution for domains (or the parameterized family of distributions $p(x|y, \nu)$) into the training procedure so that the per-

formance of $f(x)$ for the down-stream task is robust or insensitive to the uncertainty in ν.

- **parameterized models:** instead of learning a single function of the data $f(x)$, one learns a family of functions $f(x;\nu)$ that is explicitly parameterized in terms of nuisance parameters and then accounts for the dependence on the nuisance parameters in the down-stream task.
- **data augmentation:** one trains a model $f(x)$ in the usual way using training dataset from multiple domains by sampling from some distribution over ν.

In this setting it is best to consider the trained model $f(x)$ or $f(x;\nu)$ to be a fixed function and decouple the variability associated to training or the choice of architecture. The fact that one could have chosen a different architecture or learning algorithm should be treated in the same way as other choices that are made in the data analysis pipeline. While it is reasonable to want downstream inference and decisions to be robust to these choices, they are of a different nature than the uncertainty in the modeling of the data distribution. We return to this point in Sections 41.10.5 and 41.10.6.

41.10.1 *Propagation of errors*

In this Section, we consider the common scenario in which one has used some machine learning technique to train a model $f(x)$ for classification or regression and wants to assess the sensitivity of the output of $f(x)$ to uncertainty in the input x. We regard the function $f(x)$ as fixed and we are not concerned with how the model was trained.

Propagating uncertainty through a ML-based model $f(x)$ is not fundamentally different than for any other function, and one can use the standard propagation of errors formula of Sec. 39.2.1. As always, it is important to recognize the limitations of the propagation of errors formula, which is accurate when the uncertainty on x is Gaussian and the function $f(x)$ is approximately linear within the region set by the uncertainty on x.

Similarly, classifiers are often used for particle identification or event selection. In that case, one is primarily interested in the efficiency ϵ to satisfy a cut on the classifier output. The efficiency depends on the distribution $p(x|y)$ through the equation $\epsilon_y = P(f(x) > f_{\rm cut}|y) = \int H(f(x) - f_{\rm cut})p(x|y)dx$, where H is the Heaviside step function and y is an index or label for the category of data that is being considered (*e.g.*, signal vs. background or electron vs. jet). Thus, the question in this context is what is the uncertainty on the efficiency ϵ_y due to uncertainty in the distribution $p(x|y)$. In practice, the quantification of the uncertainty in the efficiency ϵ_y is usually based on either a calibration measurement on real data or estimated with simulated data. These procedures typically treat the classifier as a black-box, and thus nothing precludes using those procedures on a ML-based classifier. An early example of this approach for b-tagging can be found in Ref. [357].

In the case where simulation is used to estimate the efficiency ϵ_y and its uncertainty, one usually varies nuisance parameters ν associated to the simulation. One then uses simulated samples to approximate $\epsilon_y(\nu) = P(f(x) > f_{\rm cut}|y,\nu) = \int H(f(x) - f_{\rm cut})p(x|y,\nu)dx$. Again, the procedure for incorporating uncertainty isn't fundamentally different if the classifier $f(x)$ is based on machine learning or a hand-crafted observable.

41.10.2 *Domain adaptation*

While estimating the uncertainty for a ML-based model is not fundamentally different than any other hand-crafted observable used for regression or classification, the worry of many physicists is that by working with a high-dimensional set of features x that one is more susceptible to mis-modeling of subtle correlations. This is a valid concern, and it should be appreciated that a great deal of prior knowledge and physical insight goes into the construction of hand-crafted observables so that they will be robust to the most uncertain aspects of data. However, much of this craft is based on heuristics that are difficult to systematize. Furthermore, one can only validate that such

an observable is robust if one can explicitly evaluate the performance for a perturbed distribution. Thus in the settings where one can validate the robustness to a perturbed scenario ν_0, one must have access to $p(x|y,\nu_0)$.

One approach to formalize this type of robustness is to consider the dependence on the distribution of the output of the model $f(x)$ to the nuisance parameters. In statistics, if the distribution of f is independent of the nuisance parameters, then f is referred to as a *pivotal quantity*. This is a property that we can incorporate directly into the training procedure to target a particular notion of robustness. The authors of Ref. [358] introduced an adversarial approach (similar to what is used in the generative adversarial network of Sec. 41.3.4.2) to penalize a model during training if the distribution of the output varies with the nuisance parameters. To construct the training dataset $\{x_i, y_i, \nu_i\}_{i=1,\ldots,n}$, one must sample y and ν according to some proposal distribution (similar to a prior, but only used for the creation of training dataset, not necessarily for statistical inference), corresponding to a joint distribution $p(x,y,\nu)$. Instead of minimizing the target loss $\mathcal{L}_f$ (*e.g.* cross-entropy or squared-error) with respect to the parameters ϕ_f that parameterize the model f, one trains with a minimax strategy that also includes an adversary q with parameters ϕ_r. The trained model is characterized by the saddle point

$$\hat{\phi}_f, \hat{\phi}_r = \arg\min_{\phi_f} \arg\max_{\phi_r} E_\lambda(\phi_f, \phi_r) \,, \tag{41.87}$$

where the value function E_λ includes the target loss as well as a regularization term associated to the adversary

$$E_\lambda(\phi_f, \phi_r) = \mathcal{L}_f(\phi_f) - \lambda \mathcal{L}_r(\phi_f, \phi_r) \,. \tag{41.88}$$

The constant λ is a hyperparameter, since generally there is a tradeoff between the two terms and only in special cases can the model that minimizes $\mathcal{L}_f$ also be a pivotal quantity. The regularization term

$$\mathcal{L}_r(\phi_f, \phi_r) = \mathbb{E}_{p(x,y,\nu)}[-\log q_{\phi_r}(\nu|f_{\phi_f}(x))] \tag{41.89}$$

is an example of conditional density estimation (see Sec. 41.3.3), where the model $q_{\phi_r}(\nu|f)$ is trying to predict the distribution of the nuisance parameter ν given the output of the model $f(x)$. This term is maximized when f is independent of ν. Earlier work had also used an adversarial technique for domain adaptation, but was limited to just two domains [359–361], while here ν parametrizes a continuous family of distributions and can have multiple components corresponding to different sources of uncertainty. Furthermore, the previous work aimed to make the distribution for a high-dimensional, intermediate representation of the data be invariant to the domain shift as opposed to just the final output $f(x)$.

One way of interpreting Eq. 41.87 is that the goal is to minimize a regularized loss function $\tilde{\mathcal{L}}(\phi_f) = \arg\max_{\phi_r} E_\lambda(\phi_f, \phi_r)$, where the optimization with respect to ϕ_r is not exposed. This motivates another approach in which the regularization is not achieved through a learned adversary, but by a measure of discrepancy between distributions that can be computed directly from samples. In particular, the authors of Ref. [362] proposed the use of *distance correlation* to avoid what can be a challenging min-max optimization problem.

In either case, the optimization of the hyperparameter λ is based on the downstream task. For example, in Ref. [358] considered the case where f was a signal vs. background binary classifier where the nuisance parameter ν was associated to uncertainty in the background model. The hyperparameter λ was then optimized to maximize the approximate median significance (AMS). Similarly, the authors of Refs. [363] and [362] considered new physics searches in the context of boosted jet tagging, where the hyperparameter controls the sculpting of the side-bands used for background estimation.

While these strategies modify the training procedure so that the sensitivity to the nuisance parameters is reduced, it does not typically eliminate it. As a result, one still needs to propagate the uncertainty in the data distribution through the learned model as described in the preceding section. Furthermore, care must be taken in interpreting the loss of sensitivity to the nuisance parameter. For example, for theoretical uncertainties estimated as the difference between two different calculations (*i.e.*, two-point uncertainties), decorrelation methods may reduce the apparent uncertainty while the true uncertainty remains much larger [364].

Note, this adversarial technique has also been employed in other settings where one would like to decorrelate the output of the classifier with an observed quantity so that it can be used for background estimation [363], although other techniques like moment decomposition [365] may suffice without full decorrelation. Widely used alternative approaches to decorrelation include uboost [366], DDT [367], and using dedicated training samples that vary the chosen quantity to be decorrelated [368]. Other examples of the domain adaptation and decorrelation use cases from the Living Review include Refs. [358, 362–367, 369–378].

41.10.3 *Parameterized models*

An alternative to learning a model $f(x)$ that is pivotal—*i.e.*, whose distribution is independent of the nuisance parameter ν—is to learn a family of models $f(x;\nu)$ that is parameterized in terms of the nuisance parameters. In general, there is a tradeoff between the two terms of Eq. 41.88 for a single model $f(x)$. In a parameterized model, $f(x;\nu)$ optimizes the performance of the model for every value of ν. Parameterized classifiers were first advocated in Ref. [133] in the context of simulation-based inference (see Sec. 41.10.7) and in Ref. [379] for new physics searches. It has also been applied to simulation-based inference for effective field theory parameters in Ref. [380] and Ref. [381] provides additional pedagogical examples.

The training of a parameterized model is similar to the standard procedure. For example, if one originally wanted to minimize the squared loss function $\mathcal{L}(y, f(x)) = (y - f(x))^2$ with training dataset $\{x_i, y_i\}_{i=1,\ldots,n}$, then the corresponding training procedure for the parameterized model would be as follows. One would need to construct a training set $\{x_i, y_i, \nu_i\}_{i=1,\ldots,n}$ as described in the preceding section, construct a parameterized model $f(x;\nu)$ that takes as input the original feature vector x as well as the nuisance parameters ν, and then train using the same loss $\mathcal{L}(y, f(x;\nu)) = (y - f(x;\nu))^2$.

One complication of the parameterized approach is that it is no longer possible to evaluate the model on a dataset $\{x_i\}$ and pass on only $\{f_i\}$ for downstream analysis tasks since $f(x_i;\nu)$ still depends on ν. Instead, one delay evaluating the model to some down-stream stage when the dependence on ν would accounted for. For example, in the context of a likelihood based analysis where one is testing a hypothesis where the nuisance parameters take on a particular value ν_{test}, then one will consider the data distribution $p(x|\nu_{\text{test}})$, and at that point one would evaluate the model at the corresponding nuisance parameter value, *i.e.* $f(x;\nu_{\text{test}})$. Explicit examples are given in Refs. [133, 356, 380, 381]. While this may seem complicated, it actually corresponds to what is done in a typical likelihood-based fit when the statistical model has nuisance parameters; *i.e.* the likelihood-ratio corresponds to the model $f(x;\nu)$ as in Eq. 41.13.

41.10.4 *Data augmentation*

An intuitive approach to building in robustness to systematic effects that can lead to domain shift, is simply to augment the training dataset so that it includes examples corresponding to several values of the nuisance parameter or systematic variations. As before one can construct a dataset $\{x_i, y_i, \nu_i\}_{i=1,\ldots,n}$, but instead of leveraging the information about ν_i, one simply discards this information. This corresponds to sampling from the marginal distribution $x_i, y_i \sim p(x, y) = \int d\nu p(x, y|\nu)p(\nu)$, and is often referred to as *smearing*. One can then use this smeared dataset

for supervised learning in the traditional way. While it is possible that this approach will lead to improved robustness to systematic variations (*i.e.* generalization for ν other than the nominal value) than if systematic uncertainty weren't considered at all), this intuitive approach has several shortcomings. The approach does not yield a pivotal quantity as in the adversarial approach, so propagation of uncertainty through the network is still required. Moreover, there is no direct way to control the tradeoff between independence from the nuisance parameter and the original target loss as in the adversarial approach. Finally, it can lead to significant performance loss compared to what is possible with the parameterized approach. These tradeoffs were studied in Refs. [379, 381] with both pedagogical and physically-motivated examples.

41.10.5 *Aleatoric and epistemic uncertainty*

In the machine learning and risk assessment literature, uncertainty is often characterized in terms of *aleatoric* and *epistemic* uncertainty [382–385]. Familiarity with these terms is useful, but the distinction between the two can be ambiguous, the terms are not always consistently used, and they do not clearly map onto the concepts used physics.

For example, Ref. [384], states that "roughly speaking, aleatoric (*a.k.a.*, statistical) uncertainty refers to the notion of randomness, that is, the variability in the outcome of an experiment which is due to inherently random effects", while "epistemic (*a.k.a.*, systematic) uncertainty refers to uncertainty caused by a lack of knowledge (about the best model)." This seems clear enough, but in that same reference (and in Ref. [386]) the aleatoric uncertainty is considered irreducible, while the epistemic uncertainty could be reduced with additional information. This may seem backwards for many physicists since often in particle physics, we think of how uncertainties scale as we collect more data but keep the experimental design fixed. In that case, the statistical uncertainty will be reduced with time while the systematic uncertainty will remain constant[4]. There is no paradox here, it is simply a different point of view. The emphasis of the risk assessment community is not on collecting more data with the same experimental design, but collecting different types of data that will inform the models themselves. Clearly even for physicists, data from new experiments or calibration measurements could also reduce our systematic uncertainties. While there are exceptions in the literature, the bulk of it associates aleatoric uncertainty with the randomness of classical probability (*i.e.*, the statistical uncertainty associated to repeating the same experiment many times) and epistemic uncertainty with our state of knowledge.

Perhaps a more important distinction between the perspective of physicists and machine learning researchers has to do with the use of the term "model" and what exactly is uncertain. In physics, the systematic and epistemic uncertainty is typically associated to our understanding of the underlying physics and "the model" usually refers to the physics model, detector model encapsulated in a simulation. In contrast, for machine learning research, "the model" usually refers to the trained model $\hat{f} \in \mathcal{F}$ used as described in Section 41.2.1 (or the class of functions $\mathcal{F}$ itself). This makes sense if we recall that in the bulk of machine learning research, one has little insight into the process that generated the data (*e.g.*, images of cats and dogs, or natural language). In that sense, the epistemic uncertainty in machine learning is usually associated to uncertainty in the model parameters ϕ after training, which would be reduced if one could collect more training dataset (see Ref. [385] for this point of view).

In the literature on uncertainty quantification (UQ), which is more closely connected to physics given the role of computer simulations, the terminology is more fine grained and less ambiguous. That community uses the terms parameter uncertainty (*i.e.* nuisance parameters), structural uncertainty (*i.e.*, mismodelling), algorithmic uncertainty (*i.e.* numerical uncertainty), experimental

[4] Further complicating the relationship between the terms is that many experimental uncertainties that are characterized as systematic are actually statistical in nature as auxiliary measurements and control regions are used to constrain the corresponding nuisance parameters.

uncertainty (*i.e.*, uncertainty from experimental resolution and statistical fluctuations), and interpolation uncertainty (*i.e.*, uncertainty due to interpolating between different parameter values due to lack of computational resources).

41.10.6 *Model averaging and Bayesian machine learning*

The core of Bayesian machine learning is the model averaging view. Here one often takes a more ambitious view of learning than described in Sec. 41.2.1, which is framed mainly as function approximation. While in Sec. 41.2.1, the goal is to find a function that minimizes the risk, in Bayesian machine learning one explicitly builds a probability model $q_\phi(x, y)$ for the training dataset $\mathcal{D} = \{x_i, y_i\}_{i=1,\ldots,n}$. It is the same change in perspective that one has when one views the squared error loss function $\mathcal{L}_{\text{MSE}} = (y - f_\phi(x))^2$ as the log-likelihood for a probability model $y \sim N(f(x), \sigma)$. In addition, one assumes some prior on the model parameters $p(\phi)$, which is often a Gaussian distribution, and is analogous to Tikhonov regularization (see Sec. 41.2.5). In this way, a single trained model $\hat{f} = f_{\hat{\phi}}$ is the MAP point estimate and the more complete Bayesian solution is the entire posterior distribution over the model parameters $p(\phi|\mathcal{D})$. With this view, it is clear how increasing the number of training examples n will lead to a reduction in uncertainty on ϕ. However, this notion of epistemic uncertainty has little to do with the notion of systematic uncertainty as the term is used by particle physicists.

Bayesian methods can be applied to non-probabilistic regression problems, in which case they can provide uncertainty quantification. Consider the case of regression in traditional (non-Bayesian) machine learning. The trained model $f_{\hat{\phi}}(x)$ is used to predict the target label y. For a fixed x, the model does not provide any notion of uncertainty on the prediction. One could propagate uncertainty on x through $f(x)$, but that is also not the desired quantity to characterize the intrinsic spread $p(y|x)$ in the data, which may exist even if x has negligible uncertainty. In contrast, Gaussian process regression (a Bayesian method) does provide a natural way to communicate the uncertainty on the prediction, which is possible because one first had to specify a prior on the mean and covariance of the Gaussian process.

In the context of Bayesian deep learning and Bayesian neural networks, one would place a prior on the weights and biases of the neural network $p(\phi)$ and then use one of the many emerging techniques to calculate the approximate posterior $p(\phi|\mathcal{D})$. However, we should recognize that we have little-to-no insight into the parameters of a deep neural network, so the prior on ϕ is hardly well-justified. Furthermore, just as in all Bayesian approaches, the prior is not invariant to reparametrizing the model: $\phi \to \eta(\phi)$. While it is difficult to justify the choice of the prior on the parameters (and, thus, the resulting posterior), the resulting model may perform well empirically. In such high-dimensional parameter spaces, the bias-variance tradeoff can be dramatic.

Bayesian model averaging (BMA) performs Bayesian average over the posterior $p(\phi|\mathcal{D})$. This can be applied to any quantity f_ϕ, such as a regression or classification prediction y. Suppose we can draw from the posterior $\phi \sim p(\phi|\mathcal{D})$. For each draw we can evaluate the predicted regression variable $y = f_\phi(x) + \epsilon$, where ϵ is some noise to account for uncertainty in the predictions. We can denote this process as a draw from $p(y|x, \phi)$, $y \sim p(y|x, \phi) = N(f_\phi(x), \sigma_\epsilon^2)$, where σ_ϵ^2 is the noise variance. The BMA then performs

$$p(y|x, \mathcal{D}) = \int d\phi p(\phi|\mathcal{D}) p(y|x, \phi). \tag{41.90}$$

In practice $p(y|x, \phi)$ is evaluated by drawing samples of y and ϕ, so the posterior is defined implicitly by the samples. For example, the mean prediction is obtained by averaging $f_\phi(x)$ over the samples of ϕ, and the covariance matrix is similarly evaluated by averaging the second moments over the samples of ϕ.

Ref. [387] provides a different perspective on BMA analyzed in what are referred to as the $\mathcal{M}$-open and $\mathcal{M}$-closed settings [387]. The $\mathcal{M}$-closed setting refers to the situation where the true data generating process is in the space of models, even if it is unknown to us. In contrast, the $\mathcal{M}$-open setting refers to when the true data generating process is not in model space (*i.e.* the model is mis-specified). Interestingly, in the $\mathcal{M}$-open case one can potentially do better than any one model in the model class by considering an average over the models, since averaging can create a new model that is not in the model class. BMA provides one such averaging, but other averages, which are not weighted by $p(\phi|\mathcal{D})$, can be a better choice. When the weights of each model are optimized against appropriate loss the resulting procedure is called stacking, which has been shown to be superior to BMA in the $\mathcal{M}$-open setting [387]. Ref. [388] performed experiments indicating that in some cases model averaging can also improve predictive uncertainty estimates under domain shifts.

Neural network model averaging beyond BMA comes in several different flavors. Two successful model averaging procedures are Monte Carlo dropout [389], which uses dropout ensembling, and deep ensembles [390], which use random initialization ensembling. These methods may not only be superior to BMA, they are also often significantly faster than BMA. Whether these model averages are an approximation to BMA, or an alternative to it, remains a debated topic, and both views have been advocated. BMA itself can be accelerated using approximate methods, such as stochastic Variational Inference with reparametrization trick [391].

Finally, in the context of Bayesian model uncertainty estimation, there are two practical ways to capture: repulsive ensembles and evidential regression. Repulsive ensembles are standard deep ensembles trained with an extra diversity penalty so the ensemble members make deliberately different but plausible predictions, improving coverage with fewer models. Evidential regression uses one network to predict the parameters of a simple probabilistic family plus an "evidence" term; when data are ambiguous it learns low evidence and returns wider intervals, and when data are plentiful it narrows them. The latter has been studied for uncertainty quantification for neutrino applications [392]. In practice, both approaches can yield similarly well-calibrated uncertainties. An open direction is to bring such calibrated epistemic uncertainty into the density estimators of generative models (*e.g.*, flows, VAEs, or diffusion), for example via ensemble/Bayesian variants with diversity or evidential parameterizations, so we can represent and propagate uncertainty over whole distributions, not just point predictions.

41.10.7 *Connection to probabilistic machine learning*

We end this Section by reinforcing the connection between uncertainty quantification in traditional machine learning and the more probabilistic approaches to machine learning exemplified by simulation-based inference (see Sec 41.6) and deep generative models (see Sec. 41.3.4). In the standard approach to supervised learning (*e.g.* classification and regression) the model $f(x)$ provides a point estimate for y. Estimating an uncertainty on y goes a step further, but the complete picture would be to model the posterior distribution $p(y|x)$. Gaussian processes (see Sec. 41.8.2) are an example, but the form of the models is limited to Gaussian posteriors. In Sec 41.6 we discussed approaches to model $p(y|x)$ using conditional density estimation [69,70,128]. If we extend this task to include a family of distributions parameterized by some nuisance parameters ν, then the task is to model $p(y|x,\nu)$, which is structurally similar.

In the context of classification, the output is already probabilistic, and the interpretation of the resulting classifier is $\hat{f}_{\text{MSE}}(x) \approx p(y=1|x)$ (see Eq. 41.11). Incorporating the dependence on the nuisance parameter, then connects to the likelihood-ratio trick (see Eq. 41.13), approaches to simulation-based inference that involve learning the likelihood-ratio, and the parameterized approaches described in Sec. 41.10.3.

If one pairs the training procedure for classification, regression, or density estimation used in

the approaches above with model averaging techniques such as BMA, then it would be possible to incorporate both uncertainty associated to finite training dataset and the uncertainty associated to systematic uncertainties. However, as described in Sec. 41.10.5 and Sec. 41.10.1, it is not clear that in physics applications it is desirable to account for the variability associated to training when the more common practice is to regard the trained model $\hat{f}(x)$ as fixed.

While these probabilistic approaches to machine learning are attractive conceptually, it is known in the machine learning community that classifiers often are poorly calibrated and often overly confident in their predictions. This is a problem even if one regards the trained model $\hat{f}(x)$ as fixed. Various approaches, including model averaging, are being pursued to improve the calibration of trained models, but the problem is unlikely to be eliminated entirely. Miscalibration can be verified by evaluating the true positive and false positive rates on held out data. This is common practice in experimental particle physics, where the output of a binary classifier is rarely taken at face value. Instead, the true and false positive rates are estimated with simulated data or control samples as described in Sec. 41.10.1. Furthermore, the true and false positives can be characterized as a function of the nuisance parameters. These procedures can be used to help calibrate parameterized models based on the likelihood-ratio trick (see Refs. [133, 381]). Unfortunately, calibration in the context of density estimation is more challenging. This connects to topics and challenges in anomaly detection (see Sec. 41.3.5).

41.11 Model compression and deployment in experiments

The software and computing needs of training a machine learning model are different than those encountered when it is deployed for use. The two stages are referred to as *training* and *inference*, *i.e.* making a prediction $\hat{f}(x)$ given an input x and a trained model $\hat{f}$. Sometimes this transition also involves using different programming languages for implementing the trained model from the ones used for training them. Modern machine learning frameworks support various serialization formats to exchange trained models. For instance, `ONNX` [393] provides an open source format for many types of models, is widely supported, and can be found in many frameworks, tools, and hardware. This is important when integrating a trained model into the software frameworks used by the large experiments.

While hardware acceleration with GPUs is important for efficiently training modern machine learning techniques, there are also advantages of hardware acceleration at inference time. This may include GPUs or field programmable gate arrays (FPGAs), and the Living Review includes many example works focusing on efficient inference for a given hardware architecture [153, 180, 394–403]. 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. Recently, high-level synthesis (HLS) tools [404–406], which ingest algorithms written in C/C++ code, have lowered the barrier to entry for using FPGAs. Several tools, including hls4ml [407], FINN [408, 409], Conifer [410], and fwXmachina [411], have been developed to automatically create firmware from ML algorithms. These tools have been used for applications ranging from jet tagging [412–414] to muon transverse momentum regression [415], on-detector data compression [416], charged particle tracking [417, 418], calorimeter reconstruction [237], and anomaly detection [419–421].

For applications where latency is a key concern (*e.g.*, triggering at collider experiments), various accelerators have been investigated [237, 241, 407, 410, 411, 416, 419, 422–431]. To enable the use of an ML model in resource-constrained or latency-sensitive experimental settings, reducing the size and computational complexity of the model through *compression* is often essential. 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* [432–

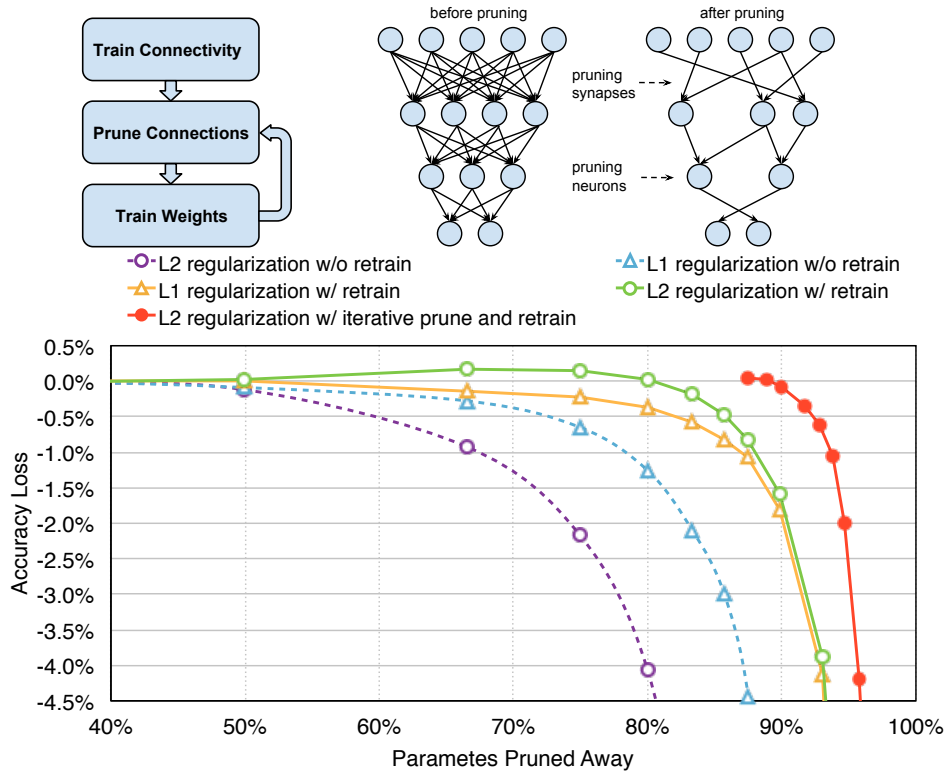

Figure 41.17: Illustration of the iterative magnitude-based parameter pruning and retraining with regularization procedure from Han et al. in NeurIPS, 2015. The top-5 accuracy loss is shown as a function of parameter reduction (sparsity) for VGG-16 on ImageNet following different pruning procedures. Without retraining, L1 regularization performs better than L2, but L2 performs better than L1 with retraining. Iterative pruning gives the best result.

448], which modifies the number of bits used to calculate and store results in the model, and *pruning* [433, 449–453], which removes model parameters. However, symbolic regression [454] and knowledge distillation [455] have also been explored to learn compact algorithms.

While it is common to use 32-bit floating-point precision, for many applications, this may not be required to ensure adequate performance. Reduced-precision formats, such as integer or fixed-point precision, may be used instead. 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 the training procedure is modified to emulate reduced precision formats. QAT results in better performance for a smaller bit width, but requires (re)training with a dedicated framework [447, 448, 456–458]. Serializing and exchanging quantized models is a challenge addressed by the `QONNX` format, which extends `ONNX` to represent arbitrary-precision quantized neural networks [459].

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. Pruning reduces the number of computations that must be performed to produce an inference result, thus reducing the hardware resources or algorithm latency. The development of pruning algorithms and understanding their behavior is an active area

of research [453]. One relatively simple method is iterative, magnitude-based pruning [407, 460], as shown in Fig. 41.17. In this process, the model is trained with L1 or L2 regularization (discussed in Sec. 41.2.5), resulting in a set of optimal parameters, where some are close to zero. Those parameters with values below a certain threshold can be set to exactly zero (thereby removing them from the model), and training can be repeated. Successive iterations of this procedure can remove more parameters until the desired reduction in parameters, or *sparsity*, is achieved. This process usually results in models that have slightly reduced performance, although the performance loss is typically negligible for sparsities $\lesssim$90% [460]. Pruning and quantization can also be applied together [429].

Finally, some solutions for deployment of ML models involve using cloud resources [461, 462] or using hardware coprocessors, like GPUs and FPGAs, *as a service* [463–466]. In this approach, coprocessor resources are decoupled from CPUs, and CPU-based clients can send inference requests to coprocessor-based servers via network calls. The advantages of this approach are coprocessors can accept inference requests from local or remote CPUs, certain types of coprocessors can be allocated for specific tasks, the coprocessor-to-CPU ratio can be optimized, the separation of software support for coprocessor and CPU workflows reduces the maintenance burden, and developments from industry can be more easily leveraged [465].

References

[1] Y. Lecun, Y. Bengio and G. Hinton, Nature **521**, 7553, 436 (2015), ISSN 14764687.

[2] J. Schmidhuber, Neural Networks **61**, 85 (2015).

[3] A. Radovic *et al.*, Nature **560**, 7716, 41 (2018).

[4] D. Guest, K. Cranmer and D. Whiteson, Ann. Rev. Nucl. Part. Sci. **68**, 161 (2018), [arXiv:1806.11484].

[5] G. Carleo *et al.*, Rev. Mod. Phys. **91**, 4, 045002 (2019), [arXiv:1903.10563].

[6] M. Feickert and B. Nachman (2021), [arXiv:2102.02770].

[7] V. Vapnik, *The nature of statistical learning theory*, Springer science & business media (2013).

[8] A. Y. Ng and M. I. Jordan, in T. G. Dietterich, S. Becker and Z. Ghahramani, editors, "Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]," 841–848, MIT Press (2001), URL https://proceedings.neurips.cc/paper/2001/hash/7b7a53e239400a13bd6be6c91c4f6c4e-Abstract.html.

[9] E. M. Metodiev, B. Nachman and J. Thaler, JHEP **10**, 174 (2017), [arXiv:1708.02949].

[10] C. Zhang *et al.*, Communications of the ACM **64**, 3, 107 (2021).

[11] P. Nakkiran *et al.*, arXiv preprint arXiv:1912.02292 (2019).

[12] T. Hastie, R. Tibshirani and J. Friedman, *The Elements of Statistical Learning*, Springer, New York, NY, USA (2001).

[13] Y. S. Abu-Mostafa, M. Magdon-Ismail and H.-T. Lin, *Learning From Data*, AMLBook (2012).

[14] M. Belkin *et al.*, Proceedings of the National Academy of Sciences **116**, 32, 15849 (2019), [arXiv:1812.11118].

[15] M. Kuusela and V. M. Panaretos, Ann. Appl. Stat. **9**, 1671 (2015), [arXiv:1505.04768].

[16] L. Rosasco, A. Tacchetti and S. Villa, CoRR **abs/1405.0042** (2014), URL http://arxiv.org/abs/1405.0042.

[17] G. E. Hinton *et al.*, CoRR **abs/1207.0580** (2012), [arXiv:1207.0580], URL http://arxiv.org/abs/1207.0580.

[18] P. Baldi and P. J. Sadowski, Advances in neural information processing systems **26**, 2814 (2013).

[19] M. Belkin, S. Ma and S. Mandal, in J. G. Dy and A. Krause, editors, "Proceedings of the 35th International Conference on Machine Learning, ICML," volume 80, 540, PMLR (2018), URL http://proceedings.mlr.press/v80/belkin18a.html.

[20] S. Gunasekar *et al.*, in J. Dy and A. Krause, editors, "Proceedings of the 35th International Conference on Machine Learning," volume 80 of *Proceedings of Machine Learning Research*, 1832–1841, PMLR (2018), URL http://proceedings.mlr.press/v80/gunasekar18a.html.

[21] L. Zdeborová, Nature Physics **16**, 6, 602 (2020).

[22] R. M. Neal, University of Toronto (1994).

[23] A. Jacot, C. Hongler and F. Gabriel, in S. Bengio *et al.*, editors, "Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada," 8580–8589 (2018), URL https://proceedings.neurips.cc/paper/2018/hash/5a4be1fa34e62bb8a6ec6b91d2462f5a-Abstract.html.

[24] Y. Bengio, A. Courville and P. Vincent, IEEE transactions on pattern analysis and machine intelligence **35**, 8, 1798 (2013).

[25] R. Acciarri *et al.* (MicroBooNE), JINST **12**, P03011 (2017), [arXiv:1611.05531].

[26] L. Dominé and K. Terao (DeepLearnPhysics), Phys. Rev. D **102**, 1, 012005 (2020), [arXiv:1903.05663].

[27] D. H. Koh *et al.* (DeepLearnPhysics) (2020), [arXiv:2007.03083].

[28] S. Farrell *et al.* (2017), URL https://dl4physicalsciences.github.io/files/nips_dlps_2017_28.pdf.

[29] S. Farrell *et al.*, in "4th International Workshop Connecting The Dots 2018 (CTD2018) Seattle, Washington, USA, March 20-22, 2018," (2018), [arXiv:1810.06111], URL http://lss.fnal.gov/archive/2018/conf/fermilab-conf-18-598-cd.pdf.

[30] F. Drielsma *et al.* (DeepLearnPhysics), Phys. Rev. D **104**, 7, 072004 (2021), [arXiv:2007.01335].

[31] X. Ju *et al.* (Exa.TrkX), Eur. Phys. J. C **81**, 10, 876 (2021), [arXiv:2103.06995].

[32] G. Dezoort *et al.* (2021), [arXiv:2103.16701].

[33] E. Parzen, The annals of mathematical statistics **33**, 3, 1065 (1962).

[34] R. A. Davis, K.-S. Lii and D. N. Politis, in "Selected Works of Murray Rosenblatt," 95–100, Springer (2011).

[35] K. S. Cranmer, Comput. Phys. Commun. **136**, 198 (2001), [hep-ex/0011057].

[36] D. P. Kingma and M. Welling, arXiv preprint arXiv:1312.6114 (2013).

[37] D. J. Rezende, S. Mohamed and D. Wierstra, in "Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014," volume 32 of *JMLR Workshop and Conference Proceedings*, 1278–1286, JMLR.org (2014), URL http://proceedings.mlr.press/v32/rezende14.html.

[38] I. J. Goodfellow *et al.*, in Z. Ghahramani *et al.*, editors, "Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada," 2672–2680 (2014), URL http://papers.nips.cc/paper/5423-generative-adversarial-nets.

[39] A. Radford, L. Metz and S. Chintala, in Y. Bengio and Y. LeCun, editors, "4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings," (2016), URL http://arxiv.org/abs/1511.06434.

[40] D. Rezende and S. Mohamed, Proceedings of the 32nd International Conference on Machine Learning **37**, 1530 (2015), URL http://proceedings.mlr.press/v37/rezende15.html.

[41] L. Dinh, D. Krueger and Y. Bengio (2015), [arXiv:1410.8516].

[42] L. Dinh, J. Sohl-Dickstein and S. Bengio, in "5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings," OpenReview.net (2017), URL https://openreview.net/forum?id=HkpbnH9lx.

[43] D. P. Kingma and P. Dhariwal, in S. Bengio *et al.*, editors, "Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada," 10236–10245 (2018).

[44] I. Kobyzev, S. Prince and M. Brubaker, IEEE Transactions on Pattern Analysis and Machine Intelligence (2020).

[45] Y. Lipman *et al.* (2023), [arXiv:2210.02747], URL https://arxiv.org/abs/2210.02747.

[46] Y. Song and S. Ermon, in H. Wallach *et al.*, editors, "Advances in Neural Information Processing Systems," volume 32, 11895, Curran Associates, Inc. (2019), URL https://proceedings.neurips.cc/paper/2019/hash/3001ef257407d5a371a96dcd947c7d93-Abstract.html.

[47] Y. Song *et al.*, CoRR **abs/2011.13456** (2020), [arXiv:2011.13456], URL https://arxiv.org/abs/2011.13456.

[48] M. S. Albergo and E. Vanden-Eijnden, in "The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023," OpenReview.net (2023), URL https://openreview.net/forum?id=li7qeBbCR1t.

[49] M. Arjovsky and L. Bottou, in "5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings," OpenReview.net (2017), URL https://openreview.net/forum?id=Hk4_qw5xe.

[50] M. Wiatrak and S. V. Albrecht, arXiv preprint arXiv:1910.00927 (2019).

[51] D. J. Rezende *et al.*, in "International Conference on Machine Learning," 8083–8092, PMLR (2020).

[52] M. C. Gemici, D. Rezende and S. Mohamed, arXiv preprint arXiv:1611.02304 (2016).

[53] J. Brehmer and K. Cranmer (2020), [arXiv:2003.13913].

[54] V. Böhm and U. Seljak, arXiv preprint arXiv:2006.05479 (2020).

[55] A. van den Oord, O. Vinyals and K. Kavukcuoglu, in I. Guyon *et al.*, editors, "Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA," 6306–6315 (2017).

[56] T. Karras *et al.*, in "6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings," OpenReview.net (2018), URL https://openreview.net/forum?id=Hk99zCeAb.

[57] T. Karras, S. Laine and T. Aila, in "IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019," 4401–4410, Computer Vision Foundation / IEEE (2019), URL http://openaccess.thecvf.com/content_CVPR_2019/html/Karras_A_Style-Based_Generator_Architecture_for_Generative_Adversarial_Networks_CVPR_2019_paper.html.

[58] M. Lucic *et al.*, in S. Bengio *et al.*, editors, "Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada," 698–707 (2018).

[59] A. A. Alemi *et al.*, in J. G. Dy and A. Krause, editors, "Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018," volume 80 of *Proceedings of Machine Learning Research*, 159–168, PMLR (2018), URL http://proceedings.mlr.press/v80/alemi18a.html.

[60] M. E. Tipping and C. M. Bishop, Journal of the Royal Statistical Society: Series B (Statistical Methodology) **61**, 3, 611 (1999), URL https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/1467-9868.00196.

[61] M. Arjovsky, S. Chintala and L. Bottou, arXiv preprint arXiv:1701.07875 (2017).

[62] L. Mescheder, S. Nowozin and A. Geiger, in I. Guyon *et al.*, editors, "Advances in Neural Information Processing Systems," volume 30, Curran Associates, Inc. (2017), URL https://proceedings.neurips.cc/paper/2017/file/4588e674d3f0faf985047d4c3f13ed0d-Paper.pdf.

[63] G. Papamakarios, I. Murray and T. Pavlakou, in "Advances in Neural Information Processing Systems," 2335–2344 (2017).

[64] C. Durkan *et al.*, in H. M. Wallach *et al.*, editors, "Advances in Neural Information Processing Systems," 7509 (2019).

[65] W. Grathwohl *et al.*, in "7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019," OpenReview.net (2019), URL https://openreview.net/forum?id=rJxgknCcK7.

[66] A. van den Oord *et al.*, arXiv:1609.03499 (2016), [arXiv:1609.03499], URL http://arxiv.org/abs/1609.03499.

[67] A. van den Oord *et al.*, in D. D. Lee *et al.*, editors, "Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain," 4790–4798 (2016), URL https://proceedings.neurips.cc/paper/2016/hash/b1301141feffabac455e1f90a7de2054-Abstract.html.

[68] B. Dai and U. Seljak, in M. Meila and T. Zhang, editors, "Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event," volume 139 of *Proceedings of Machine Learning Research*, 2352–2364, PMLR (2021), URL http://proceedings.mlr.press/v139/dai21a.html.

[69] K. Cranmer and G. Louppe, J. Brief Ideas (2016), 10.5281/zenodo.198541.

[70] G. Papamakarios and I. Murray, in "Advances in Neural Information Processing Systems," 1036–1044 (2016), ISSN 10495258, [arXiv:1605.06376].

[71] P. Holderrieth and E. Erives (2025), [arXiv:2506.02070].

[72] J. Hajer *et al.*, Phys. Rev. D **101**, 076015 (2020), URL https://link.aps.org/doi/10.1103/PhysRevD.101.076015.

[73] M. Farina, Y. Nakai and D. Shih, Phys. Rev. D **101** (2020), [arXiv:1808.08992].

[74] T. Heimel *et al.*, SciPost Phys. **6** (2019), [arXiv:1808.08979].

[75] J. Ren *et al.*, in H. M. Wallach *et al.*, editors, "Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada," 14680–14691 (2019).

[76] E. T. Nalisnick *et al.*, in "7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019," OpenReview.net (2019), URL https://openreview.net/forum?id=H1xwNhCcYm.

[77] Z. Xiao, Q. Yan and Y. Amit, arXiv preprint arXiv:2003.02977 (2020).

[78] A. Hayrapetyan *et al.* (CMS) (2025), [arXiv:2510.02168].

[79] C. Le Lan and L. Dinh, Entropy **23**, 12, 1690 (2021), ISSN 1099-4300, URL http://dx.doi.org/10.3390/e23121690.

[80] G. Kasieczka *et al.*, Phys. Rev. D **107**, 1, 015009 (2023), [arXiv:2209.06225].

[81] J. H. Collins, K. Howe and B. Nachman, Phys. Rev. Lett. **121**, 24, 241803 (2018), [arXiv:1805.02664].

[82] J. H. Collins *et al.*, The European Physical Journal C **81**, 7 (2021), ISSN 1434-6052, URL http://dx.doi.org/10.1140/epjc/s10052-021-09389-x.

[83] G. Kasieczka *et al.* (2021), [arXiv:2101.08320].

[84] T. Aarrestad *et al.* (2021), [arXiv:2105.14027].

[85] J. Devlin *et al.*, in J. Burstein, C. Doran and T. Solorio, editors, "Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)," 4171, Association for Computational Linguistics, Minneapolis, Minnesota (2019), URL https://aclanthology.org/N19-1423/.

[86] T. Brown *et al.*, in H. Larochelle *et al.*, editors, "Advances in Neural Information Processing Systems," volume 33, 1877, Curran Associates, Inc. (2020), URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[87] T. Golling *et al.*, Mach. Learn. Sci. Tech. **5**, 3, 035074 (2024), [arXiv:2401.13537].

[88] M. Leigh *et al.*, Mach. Learn. Sci. Tech. **6**, 2, 025075 (2025).

[89] J. Birk, A. Hallin and G. Kasieczka, Mach. Learn. Sci. Tech. **5**, 3, 035031 (2024), [arXiv:2403.05618].

[90] K. He *et al.*, in "CVPR," (2022), [arXiv:2111.06377].

[91] M. Caron *et al.*, in "Proceedings of the International Conference on Computer Vision (ICCV)," (2021), [arXiv:2104.14294].

[92] M. Oquab *et al.*, DINOv2: Learning Robust Visual Features without Supervision (2023).

[93] S. Wang *et al.*, in "CVPR," (2024).

[94] B. P. Duisterhof *et al.*, in "International Conference on 3D Vision 2025," (2025), URL https://openreview.net/forum?id=5uw1GRBFoT.

[95] J. Wang *et al.*, in "Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition," (2025).

[96] S. Young, Y. jae Jwa and K. Terao, Particle Trajectory Representation Learning with Masked Point Modeling (2025), [arXiv:2502.02558], URL https://arxiv.org/abs/2502.02558.

[97] Z. Hao *et al.* (2025), [arXiv:2509.07486].

[98] A. Radford *et al.*, in M. Meila and T. Zhang, editors, "Proceedings of the 38th International Conference on Machine Learning," volume 139 of *Proceedings of Machine Learning Research*, 8748, PMLR (2021), [arXiv:2103.00020], URL https://proceedings.mlr.press/v139/radford21a.html.

[99] L. Parker *et al.*, Monthly Notices of the Royal Astronomical Society **531**, 4, 4990 (2024).

[100] M. A. Hayat *et al.*, ApJ Letters **911**, 2, L33 (2021).

[101] T. Chen *et al.*, in "Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event," volume 119 of *Proceedings of Machine Learning Research*, 1597–1607, PMLR (2020), URL http://proceedings.mlr.press/v119/chen20j.html.

[102] B. M. Dillon *et al.*, SciPost Phys. **12**, 188 (2022), [arXiv:2108.04253].

[103] B. M. Dillon, R. Mastandrea and B. Nachman, Phys. Rev. D **106**, 056005 (2022), URL https://link.aps.org/doi/10.1103/PhysRevD.106.056005.

[104] B. M. Dillon *et al.*, SciPost Phys. Core **7**, 056 (2024), URL https://scipost.org/10.21468/SciPostPhysCore.7.3.056.

[105] Z. Zhao *et al.*, in "22nd International Workshop on Advanced Computing and Analysis Techniques in Physics Research," (2024), [arXiv:2408.09343].

[106] P. Harris *et al.*, Phys. Rev. D **111**, 3, 032010 (2025), [arXiv:2403.07066].

[107] P. Rieck *et al.* (2025), [arXiv:2503.11632].

[108] T. Chen *et al.*, in H. D. III and A. Singh, editors, "Proceedings of the 37th International Conference on Machine Learning," volume 119, 1597 (2020), [arXiv:2002.05709], URL https://proceedings.mlr.press/v119/chen20j.html.

[109] N. Baron Perez *et al.*, Astron. Astrophys. **699**, A302 (2025), [arXiv:2503.19111].

[110] A. Wilkinson, R. Radev and S. Alonso-Monsalve, Phys. Rev. D **111**, 9, 092011 (2025), [arXiv:2502.07724].

[111] M. Assran *et al.*, in "2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)," 15619 (2023), [arXiv:2301.08243].

[112] S. Katel *et al.*, in "7th Machine Learning and the Physical Sciences Workshop at the 38th Conference on Neural Information Processing Systems," (2024), [arXiv:2412.05333], URL https://ml4physicalsciences.github.io/2024/files/NeurIPS_ML4PS_2024_222.pdf.

[113] J. Bardhan *et al.* (2025), [arXiv:2502.03933].

[114] A. Ore, C. Heneka and T. Plehn, SciPost Phys. **18**, 5, 155 (2025), [arXiv:2410.18899].

[115] R. E. Bellman, *Dynamic Programming*, Princeton University Press, USA (1957), ISBN 069107951X.

[116] D. E. Kirk, *Optimal control theory: an introduction*, Courier Corporation (2004).

[117] K. J. Åström, Journal of Mathematical Analysis and Applications **10**, 174 (1965).

[118] J. Brehmer *et al.*, in "34th Conference on Neural Information Processing Systems," (2020), [arXiv:2011.08191].

[119] R. S. Sutton and A. G. Barto, Cambridge, MA **22447** (1998).

[120] K. Arulkumaran *et al.*, IEEE Signal Processing Magazine **34**, 6, 26 (2017).

[121] S. Carrazza and F. A. Dreyer, Phys. Rev. D **100**, 1, 014014 (2019), [arXiv:1903.09644].

[122] G. H. Mendizabal *et al.* (2025), [arXiv:2509.14894].

[123] G. N. Wojcik, S. T. Eu and L. L. Everett, Phys. Rev. D **111**, 3, 035007 (2025), [arXiv:2407.07203].

[124] H. Robbins, Bulletin of the American Mathematical Society **58**, 5, 527 (1952).

[125] J. C. Gittins, Journal of the Royal Statistical Society: Series B (Methodological) **41**, 2, 148 (1979).

[126] J. Mockus, *Bayesian approach to global optimization: theory and applications*, volume 37, Springer Science & Business Media (2012).

[127] E. Brochu, V. M. Cora and N. De Freitas (2010), [arXiv:1012.2599].

[128] K. Cranmer, J. Brehmer and G. Louppe, Proc. Nat. Acad. Sci. **117**, 48, 30055 (2020), [arXiv:1911.01429].

[129] J. Brehmer and K. Cranmer, *Artificial Intelligence for Particle Physics*, chapter Simulation-based inference methods for particle physics, World Scientific Publishing Co (2021).

[130] P. J. Diggle and R. J. Gratton, in "Journal of the Royal Statistical Society: Series B (Methodological)," volume 46, 193–212 (1984), ISSN 0035-9246.

[131] D. B. Rubin, The Annals of Statistics **12**, 4, 1151 (1984), ISSN 0090-5364, URL https://doi.org/10.1214/aos/1176346785.

[132] M. A. Beaumont, W. Zhang and D. J. Balding, Genetics **162**, 4, 2025 (2002), ISSN 00166731.

[133] K. Cranmer, J. Pavez and G. Louppe, arXiv:1506.02169 (2015), [arXiv:1506.02169], URL http://arxiv.org/abs/1506.02169.

[134] J. Brehmer *et al.*, Proc. Nat. Acad. Sci. 201915980 (2020), [arXiv:1805.12244].

[135] M. Stoye *et al.* (2018), [arXiv:1808.00973].

[136] C. Modi, F. Lanusse and U. Seljak, FlowPM: Distributed TensorFlow Implementation of the FastPM Cosmological N-body Solver (2020), [arXiv:2010.11847].

[137] J. Jasche and B. D. Wandelt, Mon. Not. Roy. Astron. Soc. **432**, 894 (2013), [arXiv:1203.3639].

[138] B. Dai and U. Seljak, Proceedings of the National Academy of Science **121**, 9, e2309624121 (2024).

[139] D. Ribli *et al.*, Monthly Notices of the Royal Astronomical Society **490**, 2, 1843 (2019).

[140] M. Arratia *et al.*, Journal of Instrumentation **17**, 1, P01024 (2022), [arXiv:2109.13243].

[141] A. Andreassen *et al.*, Phys. Rev. Lett. **124**, 18, 182001 (2020), [arXiv:1911.09107].

[142] V. Andreev *et al.* (H1) (2021), [arXiv:2108.12376].

[143] G. Aad *et al.* (ATLAS), Phys. Rev. Lett. **133**, 26, 261803 (2024), [arXiv:2405.20041].

[144] R. G. Huang *et al.*, Phys. Rev. D **112**, 1, 012008 (2025), [arXiv:2504.06857].

[145] P. T. Komiske, S. Kryhin and J. Thaler, Phys. Rev. D **106**, 9, 094021 (2022), [arXiv:2205.04459].

[146] L. Parker, A. E. Bayer and U. Seljak, Journal of Cosmology and Astroparticle Physics **2025**, 9, 039 (2025), [arXiv:2504.01092].

[147] H. Wang *et al.*, The Astrophysical Journal **794**, 1, 94 (2014), [arXiv:1407.3451].

[148] U. Seljak *et al.*, Journal of Cosmology and Astroparticle Physics **2017**, 12, 009 (2017), [arXiv:1706.06645].

[149] A. Hocker *et al.*, PoS **ACAT**, 040 (2007), [arXiv:physics/0703039].

[150] T. Mikolov *et al.*, arXiv preprint arXiv:1301.3781 (2013).

[151] E. Asgari and M. R. Mofrad, PloS one **10**, 11, e0141287 (2015).

[152] D. Guest *et al.*, Phys. Rev. **D94**, 11, 112002 (2016), [arXiv:1607.08633].

[153] T. Q. Nguyen *et al.*, Comput. Softw. Big Sci. **3**, 1, 12 (2019), [arXiv:1807.00083].

[154] E. Bols *et al.* (2020), [arXiv:2008.10519].

[155] K. Goto *et al.*, Development of a Vertex Finding Algorithm using Recurrent Neural Network (2021), [arXiv:2101.11906].

[156] R. T. de Lima (2021), [arXiv:2102.06128].

[157] Technical Report ATL-PHYS-PUB-2017-003, CERN, Geneva (2017), URL http://cdsweb.cern.ch/record/2255226.

[158] J. Pumplin, Phys. Rev. D **44**, 2025 (1991).

[159] J. Cogan *et al.*, JHEP **02**, 118 (2015), [arXiv:1407.5675].

[160] L. G. Almeida *et al.*, JHEP **07**, 086 (2015), [arXiv:1501.05968].

[161] L. de Oliveira *et al.*, JHEP **07**, 069 (2016), [arXiv:1511.05190].

[162] Technical Report ATL-PHYS-PUB-2017-017, CERN, Geneva (2017), URL http://cds.cern.ch/record/2275641.

[163] J. Lin *et al.*, JHEP **10**, 101 (2018), [arXiv:1807.10768].

[164] P. T. Komiske *et al.*, Phys. Rev. **D98**, 1, 011502 (2018), [arXiv:1801.10158].

[165] J. Barnard *et al.*, Phys. Rev. **D95**, 1, 014018 (2017), [arXiv:1609.00607].

[166] P. T. Komiske, E. M. Metodiev and M. D. Schwartz, JHEP **01**, 110 (2017), [arXiv:1612.01551].

[167] G. Kasieczka *et al.*, JHEP **05**, 006 (2017), [arXiv:1701.08784].

[168] S. Macaluso and D. Shih, JHEP **10**, 121 (2018), [arXiv:1803.00107].

[169] J. Li, T. Li and F.-Z. Xu (2020), [arXiv:2008.13529].

[170] J. Li and H. Sun (2020), [arXiv:2009.00170].

[171] J. S. H. Lee *et al.*, J. Korean Phys. Soc. **74**, 3, 219 (2019), [arXiv:2012.02531].

[172] J. Collado *et al.*, Learning to Isolate Muons (2021), [arXiv:2102.02278].

[173] Y.-L. Du, D. Pablos and K. Tywoniuk (2020), [arXiv:2012.07797].

[174] J. Filipek *et al.* (2021), [arXiv:2105.04582].

[175] Technical Report ATL-PHYS-PUB-2019-028, CERN, Geneva (2019), URL http://cds.cern.ch/record/2684070.

[176] M. Andrews *et al.* (2018), [arXiv:1807.11916].

[177] Y.-L. Chung, S.-C. Hsu and B. Nachman (2020), [arXiv:2009.05930].

[178] Y.-L. Du *et al.*, Eur. Phys. J. C **80**, 6, 516 (2020), [arXiv:1910.11530].

[179] M. Andrews *et al.* (2021), [arXiv:2104.14659].

[180] A. A. Pol *et al.* (2021), [arXiv:2105.05785].

[181] A. Aurisano *et al.*, JINST **11**, 09, P09001 (2016), [arXiv:1604.01444].

[182] R. Acciarri *et al.* (MicroBooNE), JINST **12**, 03, P03011 (2017), [arXiv:1611.05531].

[183] L. Hertel *et al.* (2017), URL https://dl4physicalsciences.github.io/files/nips_dlps_2017_7.pdf.

[184] C. Adams *et al.* (MicroBooNE), Phys. Rev. **D99**, 9, 092001 (2019), [arXiv:1808.07269].

[185] S. Aiello *et al.* (KM3NeT) (2020), [arXiv:2004.08254].

[186] C. Adams, K. Terao and T. Wongjirad (2020), [arXiv:2006.01993].

[187] L. Dominé *et al.* (DeepLearnPhysics), Phys. Rev. D **104**, 3, 032004 (2021), [arXiv:2006.14745].

[188] H. Yu *et al.*, JINST **16**, 01, P01036 (2021), [arXiv:2007.12743].

[189] F. Psihas *et al.* (2020), [arXiv:2008.01242].

[190] S. Alonso-Monsalve *et al.*, Phys. Rev. D **103**, 3, 032005 (2021), [arXiv:2009.00688].

[191] P. Abratenko *et al.* (MicroBooNE) (2020), [arXiv:2010.08653].

[192] B. Clerbaux *et al.* (2020), [arXiv:2011.08847].

[193] J. Liu *et al.* (2020), [arXiv:2012.06181].

[194] P. Abratenko *et al.* (MicroBooNE) (2020), [arXiv:2012.08513].

[195] S. Y.-C. Chen *et al.* (2020), [arXiv:2012.12177].

[196] R. Acciarri *et al.* (SBND) (2020), [arXiv:2012.01301].

[197] Z. Qian *et al.* (2021), [arXiv:2101.04839].

[198] R. Abbasi *et al.* (IceCube), A Convolutional Neural Network based Cascade Reconstruction for the IceCube Neutrino Observatory (2021), [arXiv:2101.11589].

[199] F. Drielsma *et al.*, in "34th Conference on Neural Information Processing Systems," (2021), [arXiv:2102.01033].

[200] M. Rossi and S. Vallecorsa, in "25th International Conference on Computing in High-Energy and Nuclear Physics," (2021), [arXiv:2103.01596].

[201] J. Hewes *et al.* (2021), [arXiv:2103.06233].

[202] R. Acciarri *et al.* (ArgoNeuT) (2021), [arXiv:2103.06391].

[203] V. Belavin, E. Trofimova and A. Ustyuzhanin (2021), [arXiv:2104.02040].

[204] D. Maksimović, M. Nieslony and M. Wurm (2021), [arXiv:2104.13426].

[205] A. Gavrikov and F. Ratnikov, in "25th International Conference on Computing in High-Energy and Nuclear Physics," (2021), [arXiv:2106.02907].

[206] J. García-Méndez *et al.* (ANTARES), JINST **16**, 09, C09018 (2021), [arXiv:2107.13654].

[207] K. Carloni *et al.* (2021), [arXiv:2110.10766].

[208] P. Abratenko *et al.* (MicroBooNE) (2021), [arXiv:2110.11874].

[209] D. Boyda *et al.*, Phys. Rev. D **103**, 7, 074504 (2021), [arXiv:2008.05456].

[210] G. Kanwar *et al.*, Phys. Rev. Lett. **125**, 12, 121601 (2020), [arXiv:2003.06413].

[211] P. T. Komiske, E. M. Metodiev and J. Thaler, JHEP **01**, 121 (2019), [arXiv:1810.05165].

[212] H. Qu and L. Gouskos, Phys. Rev. D **101**, 5, 056019 (2020), [arXiv:1902.08570].

[213] V. Mikuni and F. Canelli, Eur. Phys. J. Plus **135**, 6, 463 (2020), [arXiv:2001.05311].

[214] J. Shlomi *et al.*, Eur. Phys. J. C **81**, 6, 540 (2021), [arXiv:2008.02831].

[215] M. J. Dolan and A. Ore, Phys. Rev. D **103**, 7, 074022 (2021), [arXiv:2012.00964].

[216] M. J. Fenton *et al.*, Phys. Rev. D **105**, 11, 112008 (2022), [arXiv:2010.09206].

[217] J. S. H. Lee *et al.* (2020), [arXiv:2012.03542].

[218] V. Mikuni and F. Canelli, Mach. Learn. Sci. Tech. **2**, 035027 (2021), [arXiv:2102.05073].

[219] A. Shmakov *et al.*, SciPost Phys. **12**, 5, 178 (2022), [arXiv:2106.03898].

[220] C. Shimmin (2021), [arXiv:2107.02908].

[221] ATLAS Collaboration, ATLAS PUB Note ATL-PHYS-PUB-2020-014 (2020), URL https://cds.cern.ch/record/2718948.

[222] H. Qu, C. Li and S. Qian, in "Proceedings of the 39th International Conference on Machine Learning," 18281 (2022), [arXiv:2202.03772].

[223] G. Louppe *et al.*, JHEP **01**, 057 (2019), [arXiv:1702.00748].

[224] T. Cheng (2017), [arXiv:1711.02633].

[225] I. Henrion *et al.* (2017), URL https://dl4physicalsciences.github.io/files/nips_dlps_2017_29.pdf.

[226] X. Ju *et al.*, in "Machine Learning and the Physical Sciences at NeurIPS," (2020), [arXiv:2003.11603].

[227] M. Abdughani *et al.*, JHEP **08**, 055 (2019), [arXiv:1807.09088].

[228] J. Arjona Martínez *et al.*, Eur. Phys. J. Plus **134**, 7, 333 (2019), [arXiv:1810.07988].

[229] J. Ren, L. Wu and J. M. Yang, Phys. Lett. B **802**, 135198 (2020), [arXiv:1901.05627].

[230] E. A. Moreno *et al.*, Eur. Phys. J. C **80**, 1, 58 (2020), [arXiv:1908.05318].

[231] S. R. Qasim *et al.*, Eur. Phys. J. C **79**, 7, 608 (2019), [arXiv:1902.07987].

[232] A. Chakraborty, S. H. Lim and M. M. Nojiri, JHEP **19**, 135 (2020), [arXiv:1904.02092].

[233] A. Chakraborty *et al.* (2020), [arXiv:2003.11787].

[234] M. Abdughani *et al.* (2020), [arXiv:2005.11086].

[235] E. Bernreuther *et al.* (2020), [arXiv:2006.08639].

[236] J. Shlomi, P. Battaglia and J.-R. Vlimant, Machine Learning: Science and Technology **2**, 2, 021001 (2021), ISSN 2632-2153, URL http://dx.doi.org/10.1088/2632-2153/abbf9a.

[237] Y. Iiyama *et al.*, Front. Big Data **3**, 598927 (2020), [arXiv:2008.03601].

[238] X. Ju and B. Nachman, Phys. Rev. D **102**, 075014 (2020), [arXiv:2008.06064].

[239] N. Choma *et al.* (2020), [arXiv:2007.00149].

[240] Jun Guo and Jinmian Li and Tianjun Li (2020), [arXiv:2010.05464].

[241] A. Heintz *et al.*, 34th Conference on Neural Information Processing Systems (2020), [arXiv:2012.01563].

[242] Y. Verma and S. Jena (2020), [arXiv:2012.08515].

[243] F. A. Dreyer and H. Qu (2020), [arXiv:2012.08526].

[244] J. Pata *et al.* (2021), [arXiv:2101.08578].

[245] C. Biscarat *et al.*, in "25th International Conference on Computing in High-Energy and Nuclear Physics," (2021), [arXiv:2103.00916].

[246] S. Thais and G. DeZoort (2021), [arXiv:2103.06509].

[247] Y. Verma and S. Jena (2021), [arXiv:2103.14906].

[248] A. Hariri, D. Dyachkova and S. Gleyzer (2021), [arXiv:2104.01725].

[249] O. Atkinson *et al.* (2021), [arXiv:2105.07988].

[250] P. Konar, V. S. Ngairangbam and M. Spannowsky (2021), [arXiv:2109.14636].

[251] K. Cho *et al.*, in "Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)," (2014).

[252] H. Serviansky *et al.*, in H. Larochelle *et al.*, editors, "Advances in Neural Information Processing Systems," volume 33, 22080–22091, Curran Associates, Inc. (2020), URL https://proceedings.neurips.cc/paper/2020/file/fb4ab556bc42d6f0ee0f9e24ec4d1af0-Paper.pdf.

[253] C. M. Bishop, *Pattern recognition and machine learning*, springer (2006).

[254] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*, volume 2, MIT press Cambridge, MA (2006).

[255] A. Gandrakota *et al.*, JHEP **02**, 230 (2023), [arXiv:2202.05856].

[256] M. Frate *et al.* (2017), [arXiv:1709.05681].

[257] S. Mishra-Sharma and K. Cranmer, in "34th Conference on Neural Information Processing Systems," (2020), [arXiv:2010.10450].

[258] J. W. Foster *et al.*, Phys. Rev. Lett. **127**, 5, 051101 (2021), [arXiv:2102.02207].

[259] L. Breiman *et al.* (1984).

[260] XGBoosthttps://xgboost.readthedocs.io/.

[261] G. Ke *et al.*, in "Proceedings of the 31st International Conference on Neural Information Processing Systems," NIPS'17, 3149–3157, Curran Associates Inc., Red Hook, NY, USA (2017), ISBN 9781510860964.

[262] N. Erickson *et al.*, "Tabarena: A living benchmark for machine learning on tabular data," (2025), [arXiv:2506.16791].

[263] I. Narsky (2005), [arXiv:physics/0507143].

[264] G. Louppe, arXiv preprint arXiv:1407.7502 (2014).

[265] Y. Freund and R. E. Schapire, Journal of computer and system sciences **55**, 1, 119 (1997).

[266] J. H. Friedman, Annals of statistics 1189–1232 (2001).

[267] K. Fukushima, Biological Cybernetics **36**, 193 (1980).

[268] V. Nair and G. E. Hinton, in "ICML," (2010).

[269] A. L. Maas, A. Y. Hannun and A. Y. Ng, in "in ICML Workshop on Deep Learning for Audio, Speech and Language Processing," (2013).

[270] K. He *et al.*, IEEE International Conference on Computer Vision (ICCV 2015) **1502** (2015).

[271] V. Sitzmann *et al.*, in "Proc. NeurIPS," (2020).

[272] D. Hendrycks and K. Gimpel, Gaussian Error Linear Units (GELUs) (2023), [arXiv:1606.08415], URL https://arxiv.org/abs/1606.08415.

[273] P. Ramachandran, B. Zoph and Q. V. Le, Searching for Activation Functions (2017), [arXiv:1710.05941], URL https://arxiv.org/abs/1710.05941.

[274] S. Elfwing, E. Uchibe and K. Doya, Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning (2017), [arXiv:1702.03118], URL https://arxiv.org/abs/1702.03118.

[275] G. Cybenko, Mathematics of control, signals and systems **2**, 4, 303 (1989).

[276] O. Delalleau and Y. Bengio, in J. Shawe-Taylor *et al.*, editors, "Advances in Neural Information Processing Systems," volume 24, Curran Associates, Inc. (2011), URL https://proceedings.neurips.cc/paper/2011/file/8e6b42f1644ecb1327dc03ab345e618b-Paper.pdf.

[277] R. Raina, A. Madhavan and A. Y. Ng, in "Proceedings of the 26th Annual International Conference on Machine Learning," ICML '09, 873–880, Association for Computing Machinery, New York, NY, USA (2009), ISBN 9781605585161, URL https://doi.org/10.1145/1553374.1553486.

[278] Y. LeCun, Deep Learning est mort. Vive Differentiable Programming!https://www.facebook.com/yann.lecun/posts/10155003011462143 (2018), URL https://www.facebook.com/yann.lecun/posts/10155003011462143andhttps://techburst.io/deep-learning-est-mort-vive-differentiable-programming-5060d3c55074.

[279] C. Szegedy *et al.*, in "2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)," 1–9 (2015).

[280] N. Cohen and A. Shashua, CoRR **abs/1605.06743** (2016), URL http://arxiv.org/abs/1605.06743.

[281] A. Bietti, L. Venturi and J. Bruna, arXiv preprint arXiv:2106.07148 (2021).

[282] M. M. Bronstein *et al.*, arXiv preprint arXiv:2104.13478 (2021).

[283] K. Simonyan and A. Zisserman, CoRR **abs/1409.1556** (2015).

[284] S. Ren *et al.*, in C. Cortes *et al.*, editors, "Advances in Neural Information Processing Systems," volume 28, Curran Associates, Inc. (2015), URL https://proceedings.neurips.cc/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf.

[285] K. He *et al.*, in "2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)," 770–778 (2016).

[286] O. Ronneberger, P. Fischer and T. Brox, in N. Navab *et al.*, editors, "Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015," 234–241, Springer International Publishing, Cham (2015), ISBN 978-3-319-24574-4.

[287] D. Rumelhart, G. Hinton and R. Williams .

[288] Y. Bengio, P. Simard and P. Frasconi, Neural Networks, IEEE Transactions on **5**, 2, 157 (1994).

[289] R. Pascanu, T. Mikolov and Y. Bengio, in S. Dasgupta and D. McAllester, editors, "Proceedings of the 30th International Conference on Machine Learning," volume 28 of *Proceedings of Machine Learning Research*, 1310–1318, PMLR, Atlanta, Georgia, USA (2013), URL https://proceedings.mlr.press/v28/pascanu13.html.

[290] S. Hochreiter and J. Schmidhuber, Neural Computation **9**, 8, 1735 (1997).

[291] R. Socher *et al.*, in "ICML," (2011).

[292] R. Socher *et al.*, in "Proceedings of the 2011 conference on empirical methods in natural language processing," 151–161 (2011).

[293] X. Chen *et al.*, in "Proceedings of the 2015 conference on empirical methods in natural language processing," 793–798 (2015).

[294] C. Olah and S. Carter, Distill (2016), URL http://distill.pub/2016/augmented-rnns.

[295] D. Bahdanau, K. Cho and Y. Bengio (2015), 3rd International Conference on Learning Representations, ICLR 2015 ; Conference date: 07-05-2015 Through 09-05-2015.

[296] R. Bommasani *et al.*, arXiv preprint arXiv:2108.07258 (2021).

[297] P. Battaglia *et al.*, arXiv (2018), URL https://arxiv.org/pdf/1806.01261.pdf.

[298] M. Gori, G. Monfardini and F. Scarselli, in "Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.", volume 2, 729–734 vol. 2 (2005).

[299] C. R. Qi *et al.*, in "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)," (2017).

[300] C. Qi *et al.*, in "NIPS," (2017).

[301] M. Jaderberg *et al.*, in C. Cortes *et al.*, editors, "Advances in Neural Information Processing Systems," volume 28, Curran Associates, Inc. (2015), URL https://proceedings.neurips.cc/paper/2015/file/33ceb07bf4eeb3da587e268d663aba1a-Paper.pdf.

[302] M. Zaheer *et al.*, in I. Guyon *et al.*, editors, "Advances in Neural Information Processing Systems," volume 30, Curran Associates, Inc. (2017), URL https://proceedings.neurips.cc/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf.

[303] Y. Wang *et al.*, ACM Transactions on Graphics **38** (2018).

[304] X. Wang *et al.*, in "2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)," 7794–7803, IEEE Computer Society, Los Alamitos, CA, USA (2018), URL https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00813.

[305] J. Gilmer *et al.*, in D. Precup and Y. W. Teh, editors, "Proceedings of the 34th International Conference on Machine Learning," volume 70 of *Proceedings of Machine Learning Research*, 1263–1272, PMLR (2017), URL https://proceedings.mlr.press/v70/gilmer17a.html.

[306] N. Choma *et al.* (IceCube) (2018), [arXiv:1809.06166].

[307] S. R. Qasim *et al.*, The European Physical Journal C **79**, 7 (2019), ISSN 1434-6052, URL http://dx.doi.org/10.1140/epjc/s10052-019-7113-9.

[308] E. A. Moreno *et al.*, Phys. Rev. D **102**, 1, 012010 (2020), [arXiv:1909.12285].

[309] J. Pata *et al.*, Commun. Phys. **7**, 1, 124 (2024), [arXiv:2309.06782].

[310] J. Kieseler, Eur. Phys. J. C **80**, 9, 886 (2020), [arXiv:2002.03605].

[311] J. Shlomi, P. Battaglia and J.-R. Vlimant, Machine Learning: Science and Technology **2**, 2, 021001 (2021), URL https://doi.org/10.1088/2632-2153/abbf9a.

[312] A. Bogatskiy *et al.*, in H. D. III and A. Singh, editors, "Proceedings of the 37th International Conference on Machine Learning," volume 119 of *Proceedings of Machine Learning Research*, 992–1002, PMLR (2020), URL https://proceedings.mlr.press/v119/bogatskiy20a.html.

[313] S. Gong *et al.*, JHEP **07**, 030 (2022), [arXiv:2201.08187].

[314] Z. Hao *et al.*, Eur. Phys. J. C **83**, 6, 485 (2023), [arXiv:2212.07347].

[315] A. Bogatskiy *et al.* (2022), [arXiv:2211.00454].

[316] A. Bogatskiy *et al.*, JHEP **03**, 113 (2024), [arXiv:2307.16506].

[317] J. Brehmer *et al.* (2024), [arXiv:2411.00446].

[318] J. Spinner *et al.* (2024), [arXiv:2405.14806].

[319] T. S. Cohen *et al.*, arXiv e-prints (2019), [arXiv:1902.04615].

[320] F. B. Fuchs *et al.*, in "Proceedings of the 34th International Conference on Neural Information Processing Systems," NIPS '20, Curran Associates Inc., Red Hook, NY, USA (2020), ISBN 9781713829546.

[321] D. Boyda *et al.*, Phys. Rev. D **103**, 074504 (2021), URL https://link.aps.org/doi/10.1103/PhysRevD.103.074504.

[322] S. Batzner *et al.*, Nature Communications **13**, 1 (2022), ISSN 2041-1723, URL http://dx.doi.org/10.1038/s41467-022-29939-5.

[323] M. Raissi, P. Perdikaris and G. Karniadakis, Journal of Computational Physics **378**, 686 (2019), ISSN 0021-9991, URL https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[324] A. S. Krishnapriyan *et al.*, Advances in Neural Information Processing Systems **34** (2021).

[325] R. Newbury *et al.*, IEEE Access **12**, 97581 (2024), URL https://api.semanticscholar.org/CorpusID:271051266.

[326] S. Shirobokov *et al.*, in H. Larochelle *et al.*, editors, "Advances in Neural Information Processing Systems," volume 33, 14650–14662, Curran Associates, Inc. (2020), [arXiv:2002.04632], URL https://proceedings.neurips.cc/paper/2020/hash/a878dbebc902328b41dbf02aa87abb58-Abstract.html.

[327] S. Mandt, M. D. Hoffman and D. M. Blei, J. Mach. Learn. Res. **18**, 134:1 (2017), URL http://jmlr.org/papers/v18/17-214.html.

[328] Y. You, I. Gitman and B. Ginsburg (2017), [arXiv:1708.03888], URL https://arxiv.org/abs/1708.03888.

[329] I. Loshchilov and F. Hutter, Decoupled Weight Decay Regularization (2019), [arXiv:1711.05101], URL https://arxiv.org/abs/1711.05101.

[330] D. P. Kingma and J. Ba, Adam: A Method for Stochastic Optimization (2017), [arXiv:1412.6980], URL https://arxiv.org/abs/1412.6980.

[331] Y. You *et al.*, Large Batch Optimization for Deep Learning: Training BERT in 76 minutes (2020), [arXiv:1904.00962], URL https://arxiv.org/abs/1904.00962.

[332] X. Chen *et al.*, Symbolic Discovery of Optimization Algorithms (2023), [arXiv:2302.06675], URL https://arxiv.org/abs/2302.06675.

[333] S. Gasiorowski *et al.*, Machine Learning: Science and Technology **5**, 2, 025012 (2024), URL https://doi.org/10.1088/2632-2153/ad2cf0.

[334] L. Heinrich and M. Kagan, Journal of Physics: Conference Series **2438**, 1, 012137 (2023), ISSN 1742-6596, URL http://dx.doi.org/10.1088/1742-6596/2438/1/012137.

[335] M. Aehle *et al.*, Computer Physics Communications **309**, 109491 (2025), ISSN 0010-4655, URL https://www.sciencedirect.com/science/article/pii/S0010465524004144.

[336] M. Lei *et al.*, Implicit Neural Representation as a Differentiable Surrogate for Photon Propagation in a Monolithic Neutrino Detector (2022), [arXiv:2211.01505], URL https://arxiv.org/abs/2211.01505.

[337] Y. LeCun *et al.*, *Efficient backprop*, 9–48, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Verlag (2012), ISBN 9783642352881, copyright: Copyright 2021 Elsevier B.V., All rights reserved.

[338] Y. Yao, L. Rosasco and A. Caponnetto, Constructive Approximation **26**, 2, 289 (2007).

[339] L. Prechelt, in "Neural Networks: Tricks of the trade," 55–69, Springer (1998).

[340] A. Krizhevsky, I. Sutskever and G. Hinton, Neural Information Processing Systems **25** (2012).

[341] X. Glorot and Y. Bengio, in Y. W. Teh and M. Titterington, editors, "Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics," volume 9 of *Proceedings of Machine Learning Research*, 249–256, PMLR, Chia Laguna Resort, Sardinia, Italy (2010), URL http://proceedings.mlr.press/v9/glorot10a.html.

[342] S. Ioffe and C. Szegedy, in F. Bach and D. Blei, editors, "Proceedings of the 32nd International Conference on Machine Learning," volume 37, 448, PMLR (2015), [arXiv:1502.03167], URL https://proceedings.mlr.press/v37/ioffe15.html.

[343] J. L. Ba, J. R. Kiros and G. E. Hinton, Layer Normalization (2016), [arXiv:1607.06450].

[344] D. Ulyanov, A. Vedaldi and V. Lempitsky, Instance Normalization: The Missing Ingredient for Fast Stylization (2017), [arXiv:1607.08022].

[345] Y. Wu and K. He, Group Normalization (2018), [arXiv:1803.08494].

[346] T.-Y. Lin *et al.*, in D. Fleet *et al.*, editors, "Computer Vision – ECCV 2014," 740–755, Springer International Publishing, Cham (2014), ISBN 978-3-319-10602-1.

[347] O. Russakovsky *et al.*, International Journal of Computer Vision (IJCV) **115**, 3, 211 (2015).

[348] M. Cordts *et al.*, in "Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)," (2016).

[349] L. Yi *et al.*, SIGGRAPH Asia (2016).

[350] A. Radford *et al.* (2018), URL https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.

[351] T. Hellert, J. Montenegro and A. Pollastro, APL Machine Learning **2**, 4, 046105 (2024), ISSN 2770-9019, URL https://doi.org/10.1063/5.0238090.

[352] T. Hellert *et al.*, Phys. Rev. Accel. Beams **28**, 044601 (2025), URL https://link.aps.org/doi/10.1103/PhysRevAccelBeams.28.044601.

[353] C. Tan *et al.* (2023), [arXiv:2312.17016].

[354] A. Ramesh *et al.*, in M. Meila and T. Zhang, editors, "Proceedings of the 38th International Conference on Machine Learning," volume 139, 8821 (2021), [arXiv:2102.12092], URL https://proceedings.mlr.press/v139/ramesh21a.html.

[355] A. Blattmann *et al.*, Retrieval-Augmented Diffusion Models (2022), URL https://arxiv.org/abs/2204.11824.

[356] T. Dorigo and P. De Castro Manzano (2020), [arXiv:2007.09121].

[357] R. Barate *et al.* (ALEPH), Phys. Lett. B **412**, 173 (1997).

[358] G. Louppe, M. Kagan and K. Cranmer, in I. Guyon *et al.*, editors, "Advances in Neural Information Processing Systems," volume 30, Curran Associates, Inc. (2017), [arXiv:1611.01046], URL https://papers.nips.cc/paper/2017/hash/48ab2f9b45957ab574cf005eb8a76760-Abstract.html.

[359] H. Edwards and A. Storkey, arXiv preprint arXiv:1511.05897 (2015).

[360] Y. Ganin and V. Lempitsky, in F. Bach and D. Blei, editors, "Proceedings of the 32nd International Conference on Machine Learning," volume 37 of *Proceedings of Machine Learning Research*, 1180, PMLR (2015), [arXiv:1409.7495], URL https://proceedings.mlr.press/v37/ganin15.html.

[361] Y. Ganin *et al.*, J. Mach. Learn. Res. **17**, 1 (2016), [arXiv:1412.4446], URL http://jmlr.org/papers/v17/15-239.html.

[362] G. Kasieczka and D. Shih, Phys. Rev. Lett. **125**, 12, 122001 (2020), [arXiv:2001.05310].

[363] C. Shimmin *et al.*, Phys. Rev. **D96**, 7, 074034 (2017), [arXiv:1703.03507].

[364] A. Ghosh and B. Nachman, Eur. Phys. J. C **82**, 46 (2022), [arXiv:2109.08159].

[365] O. Kitouni *et al.*, JHEP **21**, 070 (2020), [arXiv:2010.09745].

[366] J. Stevens and M. Williams, JINST **8**, P12013 (2013), [arXiv:1305.7248].

[367] J. Dolen *et al.*, JHEP **05**, 156 (2016), [arXiv:1603.00027].

[368] CMS Detector Performance Summary CMS-DP-2020-002 (2020), URL https://cds.cern.ch/record/2707946.

[369] I. Moult, B. Nachman and D. Neill, JHEP **05**, 002 (2018), [arXiv:1710.06859].

[370] L. Bradshaw *et al.* (2019), [arXiv:1908.08959].

[371] ATLAS Collaboration, ATLAS PUB Note ATL-PHYS-PUB-2018-014 (2018), URL http://cds.cern.ch/record/2630973.

[372] L.-G. Xia, Nucl. Instrum. Meth. A **930**, 15 (2019), [arXiv:1810.08387].

[373] C. Englert *et al.*, Eur. Phys. J. C **79**, 1, 4 (2019), [arXiv:1807.08763].

[374] S. Wunsch *et al.*, Comput. Softw. Big Sci. **4**, 5 (2020), [arXiv:1907.11674].

[375] A. Rogozhnikov *et al.*, JINST **10**, 03, T03002 (2015), [arXiv:1410.4140].

[376] A. M. Sirunyan *et al.* (CMS), Mach. Learn. Sci. Tech. **1**, 035012 (2020), [arXiv:1912.12238].

[377] J. M. Clavijo, P. Glaysher and J. M. Katzy (2020), [arXiv:2005.00568].

[378] G. Kasieczka *et al.* (2020), [arXiv:2007.14400].

[379] P. Baldi *et al.*, Eur. Phys. J. **C76**, 5, 235 (2016), [arXiv:1601.07913].

[380] J. Brehmer *et al.*, Phys. Rev. **D98**, 5, 052004 (2018), [arXiv:1805.00020].

[381] A. Ghosh, B. Nachman and D. Whiteson (2021), [arXiv:2105.08742].

[382] W. L. Oberkampf *et al.*, Reliability Engineering & System Safety **85**, 1, 11 (2004), ISSN 0951-8320, alternative Representations of Epistemic Uncertainty, URL https://www.sciencedirect.com/science/article/pii/S0951832004000493.

[383] A. O'Hagan and J. E. Oakley, Reliability Engineering & System Safety **85**, 1, 239 (2004), ISSN 0951-8320, alternative Representations of Epistemic Uncertainty, URL https://www.sciencedirect.com/science/article/pii/S0951832004000638.

[384] E. Hüllermeier and W. Waegeman, CoRR **abs/1910.09457** (2019), URL http://arxiv.org/abs/1910.09457.

[385] A. Kendall and Y. Gal, in I. Guyon *et al.*, editors, "Advances in Neural Information Processing Systems," volume 30, Curran Associates, Inc. (2017), URL https://proceedings.neurips.cc/paper/2017/file/2650d6089a6d640c5e85b2b88265dc2b-Paper.pdf.

[386] A. D. Kiureghian and O. Ditlevsen, Structural Safety **31**, 2, 105 (2009), ISSN 0167-4730, risk Acceptance and Risk Communication, URL https://www.sciencedirect.com/science/article/pii/S0167473008000556.

[387] Y. Yao *et al.*, Bayesian Analysis **13**, 3 (2018), ISSN 1936-0975, URL http://dx.doi.org/10.1214/17-BA1091.

[388] J. Snoek *et al.*, in "Advances in Neural Information Processing Systems," volume 32, 13969 (2019), URL https://proceedings.neurips.cc/paper/2019/hash/8558cb408c1d76621371888657d2eb1d-Abstract.html.

[389] Y. Gal and Z. Ghahramani, in M. Balcan and K. Q. Weinberger, editors, "Proceedings of the 33rd International Conference on Machine Learning," volume 48, 1050, JMLR.org (2016), URL http://proceedings.mlr.press/v48/gal16.html.

[390] B. Lakshminarayanan, A. Pritzel and C. Blundell, in I. Guyon *et al.*, editors, "Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA," 6402–6413 (2017), URL https://proceedings.neurips.cc/paper/2017/hash/9ef2ed4b7fd2c810847ffa5fa85bce38-Abstract.html.

[391] D. P. Kingma, T. Salimans and M. Welling, CoRR **abs/1506.02557** (2015), URL http://arxiv.org/abs/1506.02557.

[392] D. Koh, A. Mishra and K. Terao, Journal of Instrumentation **18**, 12, P12013 (2023), URL https://doi.org/10.1088/1748-0221/18/12/P12013.

[393] J. Bai *et al.*, Open Neural Network Exchangehttps://github.com/onnx/onnx (2017), URL https://github.com/onnx/onnx.

[394] G. C. Strong (2020), [arXiv:2002.01427].

[395] V. V. Gligorov and M. Williams, JINST **8**, P02013 (2013), [arXiv:1210.6861].

[396] D. W. III *et al.* (2017), URL https://dl4physicalsciences.github.io/files/nips_dlps_2017_3.pdf.

[397] D. Bourgeois, C. Fitzpatrick and S. Stahl (2018), [arXiv:1808.00711].

[398] J. Alimena, Y. Iiyama and J. Kieseler (2020), [arXiv:2004.10744].

[399] C. Balázs *et al.* (DarkMachines High Dimensional Sampling Group) (2021), [arXiv:2101.04525].

[400] F. Rehm *et al.* (2021), [arXiv:2103.10142].

[401] C. Mahesh *et al.*, in "34th Conference on Neural Information Processing Systems," (2021), [arXiv:2104.06622].

[402] S. Amrouche *et al.* (2021), [arXiv:2105.01160].

[403] P. Goncharov *et al.*, in "24th International Scientific Conference of Young Scientists and Specialists," (2021), [arXiv:2109.08982].

[404] Xilinx, Vitis Unified Software Platform Overview (2023), URL https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[405] Intel, Intel High Level Synthesis Compiler (2023), URL https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[406] Siemens, Catapult High-Level Synthesis and Verification (2023), URL https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.

[407] J. Duarte *et al.*, JINST **13**, 07, P07027 (2018), [arXiv:1804.06913].

[408] Y. Umuroglu *et al.*, in "Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays," 65, ACM, New York, NY, USA (2017), ISBN 9781450343541, [arXiv:1612.07119].

[409] M. Blott *et al.*, ACM Trans. Reconfigurable Technol. Syst. **11**, 3 (2018), ISSN 1936-7406, [arXiv:1809.04570].

[410] S. Summers *et al.*, JINST **15**, 05, P05026 (2020), [arXiv:2002.02534].

[411] T. M. Hong *et al.*, JINST **16**, 08, P08016 (2021), [arXiv:2104.03408].

[412] E. E. Khoda *et al.*, Mach. Learn.: Sci. Technol. **4**, 2, 025004 (2023), [arXiv:2207.00559].

[413] P. Odagiu *et al.*, Mach. Learn.: Sci. Technol. **5**, 035017 (2024), [arXiv:2402.01876].

[414] CMS Collaboration, CMS Detector Performance Summary CMS-DP-2025-032 (2025), URL https://cds.cern.ch/record/2936315.

[415] CMS Collaboration, CMS Technical Design Report CERN-LHCC-2020-004. CMS-TDR-021 (2020), URL https://cds.cern.ch/record/2714892.

[416] G. Di Guglielmo *et al.* (2021), [arXiv:2105.01683].

[417] A. Elabd *et al.*, Front. Big Data **5** (2022), [arXiv:2112.02048].

[418] S.-Y. Huang *et al.*, in "33rd International Conference on Field-Programmable Logic and Applications," (2023), [arXiv:2306.11330].

[419] E. Govorkova *et al.*, Nature Mach. Intell. **4**, 154 (2022), [arXiv:2108.03986].

[420] CMS Collaboration, CMS Detector Performance Summary CMS-DP-2024-059 (2024), URL https://cds.cern.ch/record/2904695.

[421] CMS Collaboration, CMS Detector Performance Summary CMS-DP-2024-121 (2024), URL https://cds.cern.ch/record/2917884.

[422] J. Ngadiuba *et al.*, Mach. Learn.: Sci. Tech. **2**, 1, 015001 (2020), [arXiv:2003.06308].

[423] J. Krupa *et al.* (2020), [arXiv:2007.10359].

[424] L. R. M. Mohan *et al.* (2020), [arXiv:2008.09210].

[425] S. Carrazza, J. M. Cruz-Martinez and M. Rossi (2020), [arXiv:2009.06635].

[426] D. S. Rankin *et al.*, 2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC) 38 (2020), [arXiv:2010.08556].

[427] M. Rossi, S. Carrazza and J. M. Cruz-Martinez (2020), [arXiv:2012.08221].

[428] T. Aarrestad *et al.* (2021), [arXiv:2101.05108].

[429] B. Hawks *et al.* (2021), [arXiv:2102.11289].

[430] T. Teixeira, L. Andrade and J. M. de Seixas (2021), [arXiv:2103.12467].

[431] M. Migliorini *et al.* (2021), [arXiv:2105.04428].

[432] M. Nagel *et al.*, in "2019 IEEE/CVF International Conference on Computer Vision, Seoul, South Korea, October 27, 2019," 1325 (2019), [arXiv:1906.04721].

[433] S. Han, H. Mao and W. J. Dally, in Y. Bengio and Y. LeCun, editors, "4th International Conference on Learning Representations, San Juan, Puerto Rico, May 2, 2016," (2016), [arXiv:1510.00149].

[434] E. Meller *et al.*, in K. Chaudhuri and R. Salakhutdinov, editors, "Proceedings of the 36th International Conference on Machine Learning," volume 97, 4486, PMLR (2019), [arXiv:1902.01917], URL http://proceedings.mlr.press/v97/meller19a.html.

[435] R. Zhao *et al.*, in K. Chaudhuri and R. Salakhutdinov, editors, "Proceedings of the 36th International Conference on Machine Learning," volume 97, 7543, PMLR (2019), [arXiv:1901.09504], URL http://proceedings.mlr.press/v97/zhao19c.html.

[436] R. Banner *et al.*, in H. Wallach *et al.*, editors, "Advances in Neural Information Processing Systems," volume 32, 7950, Curran Associates, Inc. (2019), [arXiv:1810.05723], URL https://proceedings.neurips.cc/paper/2019/file/c0a62e133894cdce435bcb4a5df1db2d-Paper.pdf.

[437] B. Moons *et al.*, in M. B. Matthews, editor, "2017 51st Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, October 29, 2017," 1921 (2017), [arXiv:1711.00215].

[438] M. Courbariaux, Y. Bengio and J.-P. David, in C. Cortes *et al.*, editors, "Advances in Neural Information Processing Systems," volume 28, 3123, Curran Associates, Inc. (2015), URL https://proceedings.neurips.cc/paper/2015/file/3e15cc11f979ed25912dff5b0669f2cd-Paper.pdf.

[439] D. Zhang *et al.*, in V. Ferrari *et al.*, editors, "Proceedings of the European Conference on Computer Vision, Munich, Germany, September 8, 2018," 373 (2018), [arXiv:1807.10029].

[440] F. Li and B. Liu, "Ternary weight networks," (2016), [arXiv:1605.04711].

[441] S. Zhou *et al.*, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," (2016).

[442] I. Hubara *et al.*, J. Mach. Learn. Res. **18**, 187, 1 (2018), URL http://jmlr.org/papers/v18/16-456.html.

[443] M. Rastegari *et al.*, in "14th European Conference on Computer Vision (ECCV)," 525, Springer International Publishing, Cham, Switzerland (2016), [arXiv:1603.05279].

[444] P. Micikevicius *et al.*, in "6th International Conference on Learning Representations, Vancouver, BC, Canada, April 30, 2018," (2018), https://openreview.net/forum?id=r1gs9JgRZ, URL https://openreview.net/forum?id=r1gs9JgRZ.

[445] B. Zhuang *et al.*, in "2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, June 18, 2018," 7920 (2018), [arXiv:1711.00205].

[446] N. Wang *et al.*, in S. Bengio *et al.*, editors, "Advances in Neural Information Processing Systems," volume 31, 7675, Curran Associates, Inc. (2018), [arXiv:1812.08011], URL https://proceedings.neurips.cc/paper/2018/file/335d3d1cd7ef05ec77714a215134914c-Paper.pdf.

[447] Z. Dong *et al.*, in "2019 IEEE/CVF International Conference on Computer Vision, Seoul, South Korea, October 27, 2019," 293 (2019).

[448] Z. Dong *et al.*, in H. Larochelle *et al.*, editors, "Advances in Neural Information Processing Systems," volume 33, 18518, Curran Associates, Inc. (2020), [arXiv:1911.03852], URL https://proceedings.neurips.cc/paper/2020/file/d77c703536718b95308130ff2e5cf9ee-Paper.pdf.

[449] Y. LeCun, J. S. Denker and S. A. Solla, in D. S. Touretzky, editor, "Advances in Neural Information Processing Systems," volume 2, 598, Morgan-Kaufmann (1990), URL http://papers.nips.cc/paper/250-optimal-brain-damage.

[450] J. Frankle and M. Carbin, in "7th International Conference on Learning Representations," (2019), [arXiv:1803.03635], URL https://openreview.net/forum?id=rJl-b3RcF7.

[451] A. Renda, J. Frankle and M. Carbin, in "8th International Conference on Learning Representations, Addis Ababa, Ethiopia, April 26, 2020," (2020), https://openreview.net/forum?id=S1gSj0NKvB, [arXiv:2003.02389], URL https://openreview.net/forum?id=S1gSj0NKvB.

[452] H. Zhou *et al.*, in H. Wallach *et al.*, editors, "Advances in Neural Information Processing Systems," volume 32, 3597, Curran Associates, Inc. (2019), [arXiv:1905.01067], URL https://proceedings.neurips.cc/paper/2019/file/1113d7a76ffceca1bb350bfe145467c6-Paper.

[453] D. Blalock *et al.*, in I. Dhillon, D. Papailiopoulos and V. Sze, editors, "Proceedings of Machine Learning and Systems," volume 2, 129 (2020), [arXiv:2003.03033], URL https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

[454] H. F. Tsoi *et al.*, EPJ Web Conf. **295**, 09036 (2024), [arXiv:2305.04099].

[455] A. Bal *et al.*, Mach. Learn. Sci. Tech. **5**, 2, 025033 (2024), [arXiv:2311.12551].

[456] C. N. Coelho *et al.*, Nat. Mach. Intell. (2021), [arXiv:2006.10159].

[457] C. Sun *et al.* (2024), [arXiv:2405.00645].

[458] A. Pappalardo, Xilinx/brevitas (2021), URL https://github.com/Xilinx/brevitas.

[459] A. Pappalardo *et al.*, in "4th Workshop on Accelerated Machine Learning at the High-performance Embedded Architecture and Compilation 2022 Conference," (2022), [arXiv:2206.07527], URL https://accml.dcs.gla.ac.uk/papers/2022/4thAccML_paper_1(12).pdf.

[460] S. Han *et al.*, in C. Cortes *et al.*, editors, "Advances in Neural Information Processing Systems 28," volume 28, 1135, Curran Associates, Inc. (2015), [arXiv:1506.02626], URL https://papers.nips.cc/paper/2015/hash/ae0eb3eed39d2bcef4622b2499a05fe6-Abstract.html.

[461] V. Kuznetsov, L. Giommi and D. Bonacorsi (2020), [arXiv:2007.14781].

[462] O. Sunneborn Gudnadottir *et al.*, EPJ Web Conf. **251**, 02054 (2021), [arXiv:2109.00264].

[463] J. Duarte *et al.*, Comput. Softw. Big Sci. **3**, 13 (2019), [arXiv:1904.08986].

[464] M. Wang *et al.*, Front. Big Data **3**, 604083 (2021), [arXiv:2009.04509].

[465] A. Hayrapetyan *et al.* (CMS), Comput. Softw. Big Sci. **8**, 1, 17 (2024), [arXiv:2402.15366].
[466] H. Zhao *et al.*, JINST **20**, 06, P06002 (2025), [arXiv:2501.05520].