41. Machine Learning

Written November 2021 by K. Cranmer (NYU), U. Seljak (UC Berkeley; LBNL) and K. Terao (SLAC; Stanford U.).

41.1 Introduction . . . 3
41.1.1 A gentle introduction with a representative example . . . 3
41.2 Fundamental concepts . . . 4
41.2.1 Loss, risk, empirical risk . . . 4
41.2.2 Generalization . . . 5
41.3 Common tasks and their associated loss functions . . . 6
41.3.1 Supervised learning . . . 6
41.3.1.1 Regression . . . 6
41.3.1.2 A note on regularization . . . 7
41.3.1.3 Classification . . . 8
41.3.2 Unsupervised learning . . . 11
41.3.2.1 Density estimation . . . 11
41.3.2.2 Representation learning, compression, and auto-encoders . . . 12
41.3.2.3 Clustering . . . 13
41.3.3 Optimal control, reinforcement learning, and active learning . . . 14
Reinforcement learning . . . 15
Multi-arm bandits . . . 15
Bayesian optimization . . . 15
Connection to experimental design . . . 16
Active learning . . . 16
41.3.4 Anomaly detection and out-of-distribution detection . . . 17
41.3.5 Simulation-based inference . . . 18
41.3.5.1 Differentiable simulations . . . 19
41.3.5.2 Unfolding as an inverse problem . . . 20
41.4 Data representations, inductive bias, and example applications . . . 20
41.5 Flavors of ML models . . . 22
41.5.1 Support vector machines and kernel machines . . . 22
Maximum-margin classifiers . . . 23
Soft margins and slack variables . . . 23
The dual problem . . . 24
The kernel trick . . . 24
Support vector regression . . . 24
Kernel ridge regression . . . 25
Gaussian Process Regression (krigging) . . . 25
41.5.2 Decision trees . . . 26
Tree-based models . . . 26
Ensemble methods . . . 27
Bagging . . . 27
Random forests . . . 27
AdaBoost . . . 27
Gradient boosting . . . 28

41.5.3 Neural networks . . . 29
41.5.3.1 Feed-forward multi-layer perceptron . . . 29
41.5.3.2 Activation functions . . . 29
41.5.3.3 Softmax . . . 30
41.5.3.4 The rise of deep learning . . . 30
41.5.3.5 Convolutional neural networks . . . 30
41.5.3.6 Pooling . . . 32
41.5.3.7 CNN architectures for image analysis . . . 32
Region Convolutional Neural Network . . . 33
U-Net . . . 34
41.5.3.8 Residual networks and skip connections . . . 34
41.5.3.9 Recurrent neural networks . . . 36
41.5.3.10 LSTM and GRU . . . 37
41.5.3.11 Attention . . . 38
41.5.3.12 Scaled dot-product attention . . . 40
41.5.3.13 Transformer and multi-head attention . . . 41
41.5.3.14 Graph networks and geometric deep learning . . . 41
41.5.4 Deep generative models . . . 43
41.5.4.1 Variational auto-encoders . . . 44
41.5.4.2 Generative adversarial networks . . . 46
41.5.4.3 Normalizing flows, autoregressive models, and score based models . 47
41.6 Learning algorithms . . . 49
41.6.1 Gradient-based optimization . . . 49
41.6.2 Stochastic gradient descent . . . 49
41.6.3 Optimization algorithms . . . 50
41.6.4 Automatic differentiation and back propagation . . . 50
41.6.5 The vanishing and exploding gradient problems . . . 51
41.6.6 Early stopping . . . 52
41.6.7 Initialization of model parameters . . . 53
41.6.8 Input normalization . . . 53
41.6.9 Batch normalization . . . 53
41.6.10 Transfer learning: pre-training and fine-tuning . . . 54
41.6.11 Zero, one, and a few shot learning . . . 54
41.7 Incorporating uncertainty . . . 54
41.7.1 Propagation of errors . . . 55
41.7.2 Domain adaptation . . . 56
41.7.3 Parameterized models . . . 57
41.7.4 Data augmentation . . . 58
41.7.5 Aleatoric and epistemic uncertainty . . . 58
41.7.6 Model averaging and Bayesian machine learning . . . 59
41.7.7 Connection to probabilistic machine learning . . . 60
41.8 Infrastructure for deployment in experiments . . . 61

41.1 Introduction

This chapter gives an overview of the core concepts of machine learning that are relevant to particle physics with some examples of applications to the energy, intensity, cosmic, and accelerator frontiers. Machine learning (ML) is an enormous field that has grown substantially in the last decade, propelled largely 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.

Physicists are exploring and contributing to machine learning at an unprecedented rate, which poses a challenge for those that 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 downloaded here: https://github.com/iml-wg/HEPML-LivingReview. As of the time of this writing, the Living Review includes over 500 references organized hierarchically by topic. While 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.

Despite the connotations of machine learning and artificial intelligence as a mysterious and radical departure from traditional approaches, we stress that machine learning has a mathematical formulation that is closely tied to statistics, the calculus of variations, approximation theory, and optimal control theory.

The topic can be organized along a few axes, which we use to organize this section. First, there are different learning paradigms, for example supervised learning, unsupervised learning, and reinforcement learning. We focus on the first two in this review, since reinforcement learning is less commonly used in particle physics. 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 it is they are modelling (*e.g.* so-called discriminative vs. generative models), as well as the way that they are implemented (*e.g.* neural networks, decision trees, and 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, the incorporation of symmetry, *etc.*.

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 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 *feature vector* $x \in \mathbb{R}^d$. Different components of x may have different units (e.g. units of energy, momentum, position etc.). 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 used as a *label* to identify how the example was generated (*e.g.* $y = 0$ for electrons and $y = 1$ for protons). We would like to find a function $f : x \to y$ that is able to accurately *predict* the label on new data. Because we have feature-label pairs, this is considered 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.41.5.3). The goal of the *training procedure* is to find the value of the parameters ϕ

that provide the 'best' predictions, but since no model is perfect, we must be explicit about the tradeoffs. This is made concrete through a *loss function* $\mathcal{L}(f_\phi(x), y)$. For this example, instead of the obvious zero-one loss (which is 0 if $f_\phi(x)$ equals y and 1 if they are not), we choose to use the squared-loss $\mathcal{L}_{\text{sq}}(f_\phi(x), y) = (y - f_\phi(x))^2$ (which may seem *ad hoc* now, but will be motivated in Sec. 41.3.1.3). We can evaluate the average of the loss on the training set of size n, which is referred to as the *emprical risk* $\mathcal{R}_{\text{emp}}(f_\phi) = \sum_{i=1}^{n} \mathcal{L}(f_\phi(x), y)/n$. *Training* refers to numerically minimizing the empirical risk (often referred to as the *training loss* through some abuse of terminology). 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 referred to as 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*. The empirical risk evaluated on the testing set is often larger than on the training set, and large differences indicate *overfitting*, which indicates 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. Once trained, the model can be applied to data. In order to produce a binary electron vs. proton decision from the continuous ouptut of the neural network, we must threshold (*i.e.* classify as proton if $f_{\hat{\phi}}(x) > k$). The choice of the threshold k is often referred to as a working point and it sets the tradeoff between electron and proton efficiency, fake-rates, purity, etc. These familiar concepts in particle physics are usually referred to in different terms in machine learning and 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 / rejection (or equivalently the ROC curve) requires labeled data. In a particle physics context, it is recognized that the simulation is not perfect and the mismodelling is associated to the presence of systematic uncertainty. In machine learning, the discrepancy between the distribution of the training dataset and the distribution of the data that the model will be applied to in practice is referred to as *domain shift* or *distribution shift*. While mismodelling in the training dataset might lead to a less-than-optimal classifier in practice, the real source of systematic uncertainty comes from 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.

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 Fundamental concepts

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 maximizing utility or 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 $p(u)$. The *risk* for a model $f \in \mathcal{F}$ is defined as its expected loss

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

where $\mathbb{E}_p[\cdot]$ refers to the expectation with respect to the distribution p. In the context of supervised learning, the distribution $p(u)$ describes a joint distribution over the features x and the labels y

(*i.e.* $p(u) = p(x, y)$), the model only depends on the features $f(u) = f(x)$, and the loss function takes on the special form $\mathcal{L}(u, f(u)) = \mathcal{L}(y, f(x))$. In the context of unsupervised learning there are no labels, and $u = x$. 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 $p(u)$, but does have access to samples from that distribution, *i.e.* $\{u_i\}_{i=1,\dots,n}$ with $u_i \overset{i.i.d.}{\sim} p(u)$. This leads to the corresponding *empirical risk*

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

where $\hat{p}(u) = \frac{1}{n} \sum_{i=1}^{n} \delta(u - u_i)$ is referred to as the empirical distribution of the dataset $\{u_i\}_{i=1,\dots,n}$. 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 $\hat{\mathcal{F}}$ are all possible functions parametrized by the model parameters ϕ. In an idealized infinite parameter limit machine learning functions, such as neural networks, are universal approximators, such that they cover all functions and $\mathcal{F} = \hat{\mathcal{F}}$. For finite size networks this may or may not be a valid assumption. 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 equation 41.4 for too long.

While the loss function may quantify some well-motivated notion of (negative) utility, it is also common to design loss functions so that f^* has some desired property. While in practice one does not know the data distribution $p(u)$, it is constructive to imagine that one does and analyze Eq. 41.2 with the calculus of variations. In Secs. 41.3 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 from the point of view of utility. Furthermore, there are often multiple loss functions that can lead to the same f^*. Then one can think of machine learning as applied calculus of variations where one solves 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^*.

41.2.2 *Generalization*

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 due to overfitting. More concretely, for a non-negative loss function one might have $\mathcal{R}_{\text{emp}}[\hat{f}] \to 0$, while the true risk might be large ($\mathcal{R}[\hat{f}] \gg 0$). The gap between the $\mathcal{R}[\hat{f}] - \mathcal{R}_{\text{emp}}[\hat{f}]$ is typically referred to as the *excess risk*[1]. 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 testing dataset (also called validation dataset) to obtain an unbiased estimate of it. This *cross-validation* method motivates the test – train split of the data.

[1] Similarly, the gap between the true risk of the learned model and the true risk of the optimal model (*i.e.* $\mathcal{R}[\hat{f}] - \mathcal{R}[f^*]$) is referred to as the *regret*. This quantity is mainly of interest for theoretical analysis of machine learning algorithms, and not of practical concern since usually neither term is tractable.

Intuitively, a model with many parameters has more flexibility and is more prone to overfitting. A common and intuitive heuristic is that one should not fit a model with more parameters than there are data points. However, a more careful treatment reveals that this heuristic can be both pessimistic and optimistic. For example, the single-parameter model $f_\phi(x) = \text{sign}(\sin(\phi x))$ can perfectly classify any assignment of labels on data (x_i, y_i) with $x \in \mathbb{R}$ and $y \in \{-1, 1\}$ and generalize poorly. Conversely, sometimes highly over-parameterized models (that have large subspaces of their parameters where $\mathcal{R}_{\text{emp}}[\hat{f}_\phi] \to 0$) might generalize well [8, 9]. Often this is achieved through *regularization*, both explicit and implicit (section 41.3.1.2).

Structural risk minimization is a modification to the empirical risk minimization principle that was introduced by Vapnik and Chervonenkis to account for the potential for overfitting [7]. However, the bounds are based on a worst-case type analysis and are often very weak. Recall that while one cannot calculate the true risk, one can obtain an unbiased estimate of it with a held-out, independent testing sample. Thus one can empirically compare the generalization error of two models and find that one generalizes better than the other even if the bounds might suggest the opposite. One of the major conceptual shifts that happened with the rise of deep learning was to more fully appreciate that these bounds and structural risk minimization were not a good learning principle in practice and that more theoretical work is needed to close the gap between formal bounds and empirical estimates of generalization error.

41.3 Common tasks and their associated loss functions

We now move to common tasks encountered in machine learning and their associated loss functions.

41.3.1 *Supervised learning*

Supervised learning generally refers to the class of problems where the training dataset are presented as input-output pairs $\{x_i, y_i\}_{i=1,\ldots,n}$, 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 assumed that $(x_i, y_i) \overset{i.i.d.}{\sim} p(x, y)$, though $p(x, y)$ is usually not known explicitly. Finally, the loss function in supervised learning takes on the special form $\mathcal{L}(y, f(x))$. The resulting trained model is then used to predict the labels for dataset where labels are not available.

In what follows we will make use of the equalities $p(x, y) = p(x|y)p(y) = p(y|x)p(x)$ and Bayes theorem, which is discussed in Sec. 39.1. We will also use the notation $x \sim p(x)$ to indicate that the random variable x is being drawn, sampled, or generated from the distribution $p(x)$.

41.3.1.1 *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; 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 often assumed to be 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!) one has linear regression that 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. Consider

the *squared error* as a loss function, which leads to the mean-squared error (MSE) for the 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 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 that we do not discuss here.

Note that these 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. 41.5.1. In that probabilistic approach, one can still obtain a point estimator, such as the conditional expectation or the maximum a posteriori (MAP) estimator

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

and one can also derive uncertainty estimates on the predicted value y. In this setting, the prior distribution on the model family is closely related to the concept of regularization, which we touch on in Sec. in Sec. 41.3.1.2 and in Sec. 41.5.1.

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. 41.3.2.1. These two approaches are a classic examples of two different approaches to modelling. Regression with $f^*_{\text{MSE}}(x)$ is the prototypical example of *descriminative* modelling, while modelling the joint distribution is a prototypical example of *generative* modelling. Generally, discriminative approaches with supervised learning out perform generative approaches when there is sufficient data, but generative approaches can be beneficial in data-starved settings [10].

41.3.1.2 *A note on 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^*, and there is a correspondence to the issues of bias and variance discussed in Sec. 40.2 on parameter estimation. Generically, there is a bias–variance tradeoff, and when the number of parameters is large and the number of data points is not much larger, 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 for of regularization is to penalize by the L2 norm of the parameters (*i.e.* $\|\phi\|^2$), which is referred to as *Tikhonov regularization*. This appears in the form of penalized maximum likelihood, and it is also commonly used in unfolding [11]. 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 paired with linear regression this is known as *ridge regression*, and when

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

paired with kernel machines (see Sec. 41.5.1) this gives rise to kernel ridge regression or Gaussian process regression.

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 enforce translational symmetry (see Sec. 41.5.3.5 for more discussion). Similarly, one might restrict to Lipschitz continuous functions. 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 [11, 12], where we monitor the loss on training dataset and the loss on test dataset. While the training dataset loss continues to decrease with more gradient descent cycles, the test loss may not, and early stopping stops the training when test loss flattens out or begins to increase. Another powerful form of regularization used in deep learning models is known as *dropout* [13], which randomly removes some some parts of the model during training and can be thought of as implementing a type of model averaging [14]. What is more surprising is that in the case of highly over-parameterized models where there is a large degenerate parameter space that achieves zero loss, $\Phi_0 = \{\phi | \mathcal{R}_{\text{emp}}[f_\phi]\} = 0$, that the dynamics of the optimization algorithm that is used will break the degeneracy and favor some particular $\hat{\phi} \in \Phi_0$ as if an additional regularization term was secretly included. Despite zero loss and over-parametrization, the corresponding generalization error may be small, a phenomenon called *benign overfitting* [15]. Importantly, the dynamics of different optimization algorithms will have different implicit regularization effects, and thus favor different parameter points in Φ_0 that will have different generalization error [16]. Understanding this interaction is a topic of contemporary research in machine learning [17].

41.3.1.3 *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 a model $g : \mathcal{X} \to \mathbb{R}^{|\mathcal{Y}|}$ that predicts continuous probabilities for each class (*i.e.* $g(x) \approx p(y|x)$) followed by an operation that then chooses the discrete label $y \in \mathcal{Y}$, such as a fixed threshold or $f(x) = \arg\max g(x) \approx \arg\max_y p(y|x)$. 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 mean-squared error $\mathcal{L}_{\text{MSE}}(y, f(x))$ in Eq. 41.6 for the loss function. The resulting model will

approximate f^*_{MSE}, the conditional expectation of Eq. 41.7 takes on the form

$$f^*_{\text{MSE}}(x) = \mathbb{E}_{p(y|x)}[y] \to p(y=1|x) = \frac{p(x|y=1)p(y=1)}{p(x|y=0)p(y=0) + p(x|y=1)p(y=1)} \,. \tag{41.10}$$

That is the MSE loss for binary classification leads to the Bayesian posterior probability that the label $y = 1$ given the feature vector x.

Equation 41.10 highlights an important general feature of supervised learning relevant for particle physics, which is that the joint distribution $p(x, y)$ of the training dataset implies a prior distribution $p(y)$ on the labels or classes. This prior distribution need not reflect a degree of belief or the frequency in real data, it represents the frequency in the training dataset. However, it is important to keep in mind that 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 that the probabilistic interpretation of the result will not be properly calibrated. 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 the 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. Adapting to the notation of this chapter, we have

$$f^*_{\text{N.P.}}(x) = \frac{p(x|y=1)}{p(x|y=0)} \,, \tag{41.11}$$

which does not depend on the prior probabilities $p'(y = 0)$ or $p'(y = 1)$ as in Eq. 41.10, or, equivalently, assumes equal priors $p'(y = 0) = p'(y = 1)$. From Bayes theorem, we have the identity $p(y = 1|x)/p(y = 0|x) = [p(x|y = 1)p(y = 1)]/[p(x|y = 0)p(y = 0)]$, which can be used to show that the two functions are related by a one-to-one, monotonic transformation

$$f^*_{\text{N.P.}}(x) = \frac{p(y=0)}{p(y=1)} \frac{f^*_{\text{MSE}}(x)}{1 - f^*_{\text{MSE}}(x)} \,, \tag{41.12}$$

which is referred to as the *likelihood-ratio trick*, which plays an important role in simulation-based inference (see Sec. 41.3.5). Importantly, the monotonic transformation does not impact the tradeoff of type-I and type-II error (or, equivalently, the FPR and TPR), therefore the ROC curve for $f_{\text{N.P.}}(x)$ and $f_{\text{MSE}}(x)$ are identical and do not depend on the prior probabilities $p(y)$. This property has been leveraged in the context of *weakly supervised* approaches [18] and enables one to train a classifier on data without access to labels as long as one has two datasets with different ratios $p(y = 1)/p(y = 0)$ and the same conditional distribution $p(x|y)$ of the features given the labels.

A generalization of the binary loss function for classification of Eq. 41.10, which applies to multiple classes, is the *cross-entropy* loss

$$\mathcal{L}_{\text{xe}}(y, f(x)) = -\sum_{c \in |\mathcal{Y}|} \mathbf{1}(y = c) \log(f_c(x)) \,, \tag{41.13}$$

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 from maximizing the posterior of Eq. 41.57 using a discrete set

of class labels y, which identifies $f_c(x) = \tilde{f}(y = c|x) = p(y = c|x)$ and thus enforces the constraint $\sum_c f_c(x) = 1$ and $f_c(x) \geq 0$ (for all $x \in \mathcal{X}$, *e.g.* by using the *softmax* function). The notation is aligned with the interpretation of $\tilde{f}(y|x)$ 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

$$\mathcal{R}_{\text{xe}}[f] = \mathbb{E}_{p(x,y)}\left[-\sum_{c\in|\mathcal{Y}|} \mathbf{1}(y = c)\log f_c(x)\right] = -\sum_{c\in|\mathcal{Y}|} p(y = c)\mathbb{E}_{p(x|y)}[\log \tilde{f}(y = c|x)] \,. \tag{41.14}$$

This is equivalent to $\mathcal{R}_{\text{xe}}[f] = \mathbb{E}_{p(x)}[H[p(y|x), \tilde{f}(y|x)]]$, where $H[p, f] = \mathbb{E}_p[-\log f]$ is the cross-entropy between the two distributions. One can use a Lagrange multiplier λ to enforce the normalization constraint (e.g. equation 41.10) and the calculus of variations to show that

$$f^*_{\text{x.e.},c}(x) = \lambda p(x, y = c) = \lambda p(y = c|x)p(x) = p(y = c|x) \,. \tag{41.15}$$

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 $H[p, f_\phi]$ with respect to ϕ is equivalent to minimizing the forward KL divergence

$$KL(p||f_\phi) \coloneqq \mathbb{E}_p[\log p(x)) - \log f_\phi] = H[p, f_\phi] - H[p] \,, \tag{41.16}$$

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

One can also consider the reverse KL divergence $KL[f_\phi||p]$, which is also minimized by $f_\phi = p$; however, this requires one to be able to generate samples $x_i \sim f_\phi(x)$ and be able to evaluate the probability density $p(x_i)$. Often this is not the case for real world data, but this approach is useful in the context of variational inference, where $f_\phi(x)$ is an approximation to the posterior of x, and $p(x)$ is the likelihood times the prior, which can be evaluated on samples x_i drawn from f_ϕ. Because likelihood times prior is not normalized this KL divergence optimizes the lower bound to the normalization (Evidence Lower Bound Objective or ELBO).

The forward KL is also closely related to the variational free energy principle in statistical mechanics where $H[f_\phi]$ represents the entropy of the variational distribution, $p(x) \propto \exp(-E(x)/kT)$ is the Boltzman factor for the state x with energy $E(x)$, and $H[f_\phi, p] = \mathbb{E}_{f_\phi}[E(x)]$ represents the expected energy for the variational distribution. As is well known to physicists, minimizing the free energy involves a balance between minimizing the energy and maximizing the entropy.

In some cases it is possible to augment the training dataset with an unbiased, stochastic estimate of $p(y = c|x)$ that we denote $s_c(x, z)$. For example, when the simulation involves latent variables z (*i.e.* Monte Carlo truth quantities), then the simulation encodes $p(x, z|y)$, the joint distribution over the observed features x and the latent variables z conditioned on the class y. In many cases the simulation evolves through a Markov process (*e.g.* the detector response only depends on the momenta of the incoming particles z, not the details of the hard scattering y). In that case, it is often possible to calculate $s_c(x, z) = p(y = c|x, z)$ for each training sample [19]. Using the identity, $\mathbb{E}_{p(z|x)}[p(y|x, z)] = p(y|x)$, we see that $s_c(x, z)$ is an unbiased estimator of $p(y = c|x)$. In this case, one can use $s_c(x, z)$ in place of the indicator function in Eq. 41.13 to construct an improved (lower-variance) loss function that reproduces the same cross-entropy risk function[3,4]

$$\mathcal{R}'_{\text{xe}}[f] = \mathbb{E}_{p(x,y,z)}\left[-\sum_{c\in|\mathcal{Y}|} s_c(x, z)\log f_c(x)\right] = -\sum_{c\in|\mathcal{Y}|} \mathbb{E}_{p(x)}[p(y = c|x)\log f_c(x)] \,, \tag{41.17}$$

[3] A similar approach can also be used for the squared-error, see Ref. [20].

[4] The right hand side of Eqs. 41.14 and 41.17 are written in a different form, but are equivalent.

which yields the same optimal classifier $f^*_{\text{x.e.},c}(x) = p(y = c|x)$ [20, 21].

We note that 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.

41.3.2 *Unsupervised learning*

Unsupervised learning generally refers to the class of problems that use unlabeled training dataset $\{x_i\}_{i=1,\ldots,n}$, where $x_i \in \mathcal{X}$ are the input features. Furthermore, it is typically assumed that $(x_i) \overset{i.i.d.}{\sim} p(x)$, though $p(x)$ is usually not known explicitly. Finally, the loss function in supervised learning takes on the special form $\mathcal{L}(x, f(x))$.

A related concept is that of self-supervised learning, which also aims to distill useful features in the data without requiring supervision labels for every sample in the input data. Self-supervised methods can make use of large unsupervised datasets and build meaningful representations by performing data augmentation, and learning the latent space mapping that is insensitive to it. For example, in the case of galaxy images one may augment the data by performing image rotations, adding noise, size scaling, adding point spread function smoothing etc., all of which are realistic transformations expected in a real galaxy image survey [22]. Self-supervised learning then learns the latent space representation where all of these augmentations of the same training sample result in the same latent space position. This training is augmented with contrastive learning, which ensures that different training samples do not all collapse to the same latent space position. When this latent space representation is used for downstream tasks such as classification it outperforms other forms of supervised learning [23].

41.3.2.1 *Density estimation*

The goal of density estimation is to estimate a distribution $p(x)$ based on samples $\{x_i\}_{i=1,\ldots,n}$ with $x_i \overset{i.i.d.}{\sim} p(x)$. 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:

$$\mathcal{R}_{\text{emp,xe}}[f_\phi] = -\frac{1}{n}\sum_{i=1}^{n} \log f_\phi(x) \,, \tag{41.18}$$

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

$$\mathcal{R}_{\text{xe}}[f_\phi] = \mathbb{E}_{p(x)}[-\log f_\phi(x)] \,, \tag{41.19}$$

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 $f^*_{\text{xe}}(x) = p(x)$.

The concepts of generalization and overfitting are particularly acute in unsupervised learning, where the likelihood maximization of equation 41.18, 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 $\{x_i\}_{i=1,\ldots,n}$. 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 [24–26].

41.3.2.2 *Representation learning, compression, and auto-encoders*

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 language of classical statistics). There is a spectrum of representations with trade-offs. At one end of this spectrum is lossless compression that allows you 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 down-stream 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, angles between particles, etc. This type of feature engineering generally improved performance for shallow neural networks, decision trees, *etc.*; however, with the rise of deep learning this is often no longer necessary and often limits performance compared to working with the original features. 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. [27].

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,

$$O : \mathbb{R}^k \to \mathbb{R}^d, z \mapsto Oz,\ OO^T{=}I_d \tag{41.20}$$

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),

$$\mathcal{L}_{a.e.}(x, f(x)) = \|x - f(x)\|^2 \,. \tag{41.21}$$

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 $\sigma^2_{\text{recon}} = \sum_{i=k+1}^{d} \lambda_i$.

Another common type of representation learning and nonlinear dimensionality reduction is based on the *auto-encoder* $f = g \circ e : \mathcal{X} \to \mathcal{X}$, 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 auto-encoder. 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. 41.21 serves as a natural loss function of an auto-encoder.

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 auto-encoders 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. Secondly, it may not be robust to domain shift (see Sec. 41.7.2). One approach to address these issues is the *denoising auto-encoder*, which uses a form of regularization that corrupts the input with noise $x' \sim q(x'|x)$ while still targeting reconstruction of the uncorrupted input x.

$$\mathcal{L}_{d.a.e.}(x, f(x)) = \|x - f(x')\|^2 \qquad \text{with} \qquad x' \sim q(x'|x)\ , \tag{41.22}$$

where $q(x'|x)$ is some probability distribution such as a multivariate normal.

41.3.2.3 *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.

While k-means and many other clustering algorithms are defined in terms of an optimization problem, the optimization objective is often not representative of the actual notions of performance in a given context. As a result, there are a number of quantities used for the evaluation and assessment of the resulting clustering. These include Davies–Bouldin index, Dunn index, Rand index, Jaccard index, purity, F-measure, Hopkins statistic, *etc.* [28].

41.3.3 *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. There is a tremendous amount of literature on the subject, and it is closely connected to planning, dynamic programming, and reinforcement learning. Optimal control can be seen as an extension of the calculus of variations, and thus generalizes the framing of learning presented in Sec. 41.2.

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. We follow common notation, such as that found in Wikipedia.

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:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\right], \tag{41.24}$$

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" [29]. For continuous-time systems, the optimal policy is defined by the Hamilton–Jacobi–Bellman equation [30].

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 [31].

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 [32]. 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. For example, in Q-learning one uses experience to estimate the array $Q(s, a)$ defined as

$$Q(s, a) = \sum_{s'} P_a(s, s')(R_a(s, s') + \gamma V(s')). \tag{41.25}$$

While this function is also unknown, it can be estimated during learning based on (s, a) pairs together with the outcome s'.

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. [33] for an introduction and Ref. [34] for a recent review.

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 trade-off 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 [35] and Gittins who derived an optimal policy under some conditions [36].

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 41.5.1) are often used to model the distribution over $f(x)$. The optimization techniques that use this framing are generically referred to as Bayesian Optimization [37].

Optimization in this context is usually characterized by an *exploration-exploitation* trade-off, 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 $\{f(x_t)\}_{t=1,\dots,T}$. 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 [38].

Connection to experimental design In the context of physics experiments we often want to build the next generation experiment which can reach certain target precision on parameters of interest that the experiment can measure. To achieve this precision we may deploy the concept of *experimental design*, where we the objective is to optimize some objective that quantifies the expected performance of an experiment. Moreover, we would like to achieve this within some constraints such as a minimal cost. Such problems are often solved using the Fisher matrix optimization, where Fisher matrix can be viewed as the expectation of the inverse covariance matrix. We can define

$$t(x|\theta_0) \coloneqq \nabla_\theta \log p(x|\theta)|_{\theta_0} \,, \tag{41.26}$$

where $t(x|\theta_0)$ is referred to by statisticians as the score [20]. The score plays an important role in the classical statistics as it is a sufficient statistic when $p(x|\theta)$ is in the exponential family, and through the Rao-Cramér-Fréchet bound on the variance of an estimator for θ, and is used to define the Fisher information matrix $I_{ij}(\theta) \coloneqq \mathbb{E}_{p(x|\theta)}[t_i(x|\theta)t_j(x|\theta)]$. The Fisher information, in turn is an important object in experimental design. In particle physics, the score is closely related to the concept of optimal observables. The corresponding diagonal element of the inverse of the Fisher matrix thus provides expected uncertainty estimation of a given parameter, and is a lower bound to the error of an unbiased estimator (Cramér-Rao bound). In experimental design we vary the experiment parameters within the constraints such as the total cost to minimize this uncertainty. This framework has been widely used in cosmology, where Fisher analysis is the foundation of any experiment proposal.

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 $\{y_t = f(x_t)\}_{t=1,\dots,T}$; 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 labelled 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 unlabelled 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 labelling required to train an ML system (and therefore cost and time).

41.3.4 *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 auto-encoders a popular technique is to use the reconstruction error of Eq. 41.21 to identify an outlier as one with a large reconstruction error [39, 40]. 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 auto-encoder 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 test 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 41.5.4.3) 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 [41]: for example, a zero variance pixels leads to an infinite $p(x)$, and noise must be added to regularize it. But 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. A related issue is that of typicality: an in-distribution likelihood will typically be lower than the maximum value. For a Gaussian likelihood the typical set is on average be $n/2$ in log likelihood below the peak value, where n is the dimensionality of the data. So an out-of-distribution dataset that is closer to the peak would be preferred in terms of likelihood even though its distribution does not match the in-distribution data. If this happens by chance on low variance directions which dominate the likelihood, normalizing flows can assign higher likelihoods to out-of-distribution data than to in-distribution training dataset [42]. A number of techniques have been proposed to circumvent these limitations, such as likelihood regret [43], likelihood-ratio [41], likelihood in auto-encoder latent space [44], and Wasserstein distance training of the likelihood $p(x)$ [45]. These methods can achieve better anomaly detection performance than the auto-encoder reconstruction error [44, 46].

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 [47]. All these approaches are largely complementary to each other [48]. An example of different anomaly detection methods applied to HEP is LHC Olympics 2020 and Dark Machines challenges [49, 50].

41.3.5 *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 [51, 52]. 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, 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, and z_d associated to the interaction of particles with the detector (see Sec. 34). As such, the full simulation chain can be expressed symbolically as

$$p(x|\theta) = \int dz p(x,z|\theta) = \int \mathrm{d}z_d \int \mathrm{d}z_s \int \mathrm{d}z_p \, p(x|z_d)p(z_d|z_s)p(z_s|z_p)p(z_p|\theta) \, , \tag{41.27}$$

where θ are the Lagrangian parameters that dictate the hard scattering. Evaluating the likelihood is intractable as it would require evaluating the integral above for each event.

While the likelihood is intractable, simulators provide the ability to generate synthetic data $x_i \overset{i.i.d.}{\sim} p(x|\theta)$ for any value of the parameters θ. One can use a suitable proposal distribution $\tilde{p}(\theta)$, sample $\theta_i \overset{i.i.d.}{\sim} \tilde{p}(\theta)$, 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[5].

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 [53]. Alternatively, one can use an explicit parametric family (such as a falling exponential, a Gaussian distribution, *etc.*) 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 [54, 55]. 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. 41.5.4.3) and the loss functions for density estimation (see Sec. 41.3.2.1) to learn a surrogate model for the likelihood $\hat{f}(x|\theta) \approx p(x|\theta)$ [56]. Similarly, one can use conditional density estimation to learn a surrogate model for the posterior

[5] 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.

$\hat{f}(\theta|x) \approx p(\theta|x)$, which may involve including the prior-to-proposal ratio $\tilde{p}(\theta)/p(\theta)$ [57]. 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. 41.12 [20, 58].

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

$$r(x_i, z_i|\theta_0, \theta_1) \coloneqq p(x_i, z_i|\theta_0)/p(x_i, z_i|\theta_1) \ , \tag{41.28}$$

which can be used to reduce the variance for the squared-error loss or the improved cross-entropy estimator of Eq. 41.17 [20, 21]. 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 addition, one can often augment the training dataset with the joint score

$$t(x_i, z_i|\theta_0) \coloneqq \nabla_\theta \log p(x_i, z_i|\theta)|_{\theta_0} \ . \tag{41.29}$$

Regressing on the joint score with the squared loss function $\mathcal{L}(t(x, z|\theta_0), f(x)) = (t(x, z|\theta_0) - f(x))^2$ corresponds to risk functional

$$\mathcal{R}_{\text{score}}[f] \coloneq \mathbb{E}_{p(x,z|\theta_0)}[(\nabla_\theta \log p(x_i, z_i|\theta)|_{\theta_0} - f(x))^2] \ . \tag{41.30}$$

One can show with the calculus of variations that the risk is minimized by the score of equation 41.26.

41.3.5.1 *Differentiable simulations*

One of the approaches to make inference 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. Latent variable models are known in Machine Learning as generative models and are discussed below, but latent variable models are also common in scientific applications. As an example, in cosmology one can view the initial conditions of the large scale structure or cosmic microwave background as Gaussian distributed latent variables.

While differentiable simulations have traditionally not been developed for scientific applications, the success of Machine Learning, where backpropagation algorithm combined with gradient descent optimization (see Sec. 41.6.1) is the basis of its recent advances, has inspired a new look at this. One recent example is FlowPM cosmological N-body simulation, which takes advantage of Mesh-Tensorflow to achieve a GPU-accelerated, distributed, and differentiable simulation [59].

One broad class of inference problems where gradients make the problem simpler are the so called inverse problems. As a simple example let's assume the data x are observed with some Gaussian noise with known variance σ^2, equal for all data points. The likelihood of the data can be written as

$$\log p(x|z, \theta) = -\frac{N}{2} \log[2\pi\sigma^2] - \frac{||x - g(z, \theta)||^2}{2\sigma^2}. \tag{41.31}$$

Here $g(z, \theta)$ is the forward model (generator or decoder), i.e. the simulation output in the absence of noise. The joint distribution is $p(x, z|\theta) = p(x|z, \theta)p(z|\theta)$, where $p(z|\theta)$ is the prior of latent space variables, assumed to be known, with a known gradient with respect to θ and z. In this case the gradients $\nabla_\theta \log p(x, z|\theta)$ and $\nabla_z \log p(x, z|\theta)$ are available when the simulation forward model gradients $\nabla_\theta g(z, \theta)$ and $\nabla_z g(z, \theta)$ are available.

Availability of simulation gradients in turn enables gradient based optimization or sampling. In sampling approach one can use Monte Carlo Markov Chains to obtain the posterior samples of latent space z and parameter space θ [60]. In optimization approach one can find the best-fit point $\hat{z}$, which is the MAP estimate of the latent space variables z given x and fixed θ [61]. Another optimization approach is Variational Inference, which attempts to model the posterior probability distribution using optimization. This is discussed further in Section 41.5.4.

41.3.5.2 *Unfolding as an inverse problem*

While much of the work on simulation-based inference described above is aimed at inferring the parameters θ of the simulator, there is also work that aims to to infer the latent variables z. Here it is useful 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. Examples of recent work exploring these approaches are Refs. [62–73]. In addition, there has been work to infer the posterior distribution for an individual event $p(z|x_i)$ using probabilistic programming techniques [74–76].

41.4 Data representations, inductive bias, and example applications

In Sec. 41.3 we describe the input data as living in an abstract space $x_i \in \mathcal{X}$. In this section, we briefly describe 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 will describe 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, multi-layer perceptrons, support vector machines, and tree-based methods found in popular tools such as `TMVA` [77].

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` [78]. 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 embedings 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 [79].

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 t-th electron in the i-the 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. 41.5.3.9) are particularly well suited to sequential data. Examples applications from the Living Review include [80–85].

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 p-th 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. 41.5.3.5, are particularly well suited to image-like data. Example applications from the Living Review include [81, 86–108].

It is also possible that the 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 [109–139]. 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. [140, 141]).

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 neural networks and convolutional neural networks are good examples of what is called *inductive bias* as the models do incorporate something about the structure of the data that they are expected to be used with. 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 really are 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 invariance*. A standard sequential or convolutional model will not generally be permutation invariant, but recently 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 [100, 142–152]

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 [153,154]. 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 [121,128,131,132,134,155–181].

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 embeded 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 [182] or to learn a function that takes sets as input and produces graphs as output as in the Set2Graph mode [183]. 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 [140,141]. The development of the necessary modelling components to enable practitioners to compose and train these types of models is a significant development for the field of physics.

41.5 Flavors of ML models

41.5.1 *Support vector machines and kernel machines*

Support Vector Machines (SVM) 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. While their history goes back to Vapnik and Chervonenkis in the 1960s, they gained popularity after Bernhard Boser, Isabelle Guyon and Vladimir Vapnik suggested a way to create nonlinear classifiers by applying the kernel trick in 1992 [184]. Originally they were developed for binary classification and only supported therestricted case where the training dataset can be separated without errors, but in 1995 Cortes and Vapnik extended the technique to non-separable training dataset [185]. A variant targeting regression, known as support vector regression (SVR), was developed in 1997 [186]. The theory around support vector machines was well developed, and they dominated machine learning research for roughly a decade before the rise of deep learning (see Sec. 41.5.3.4).

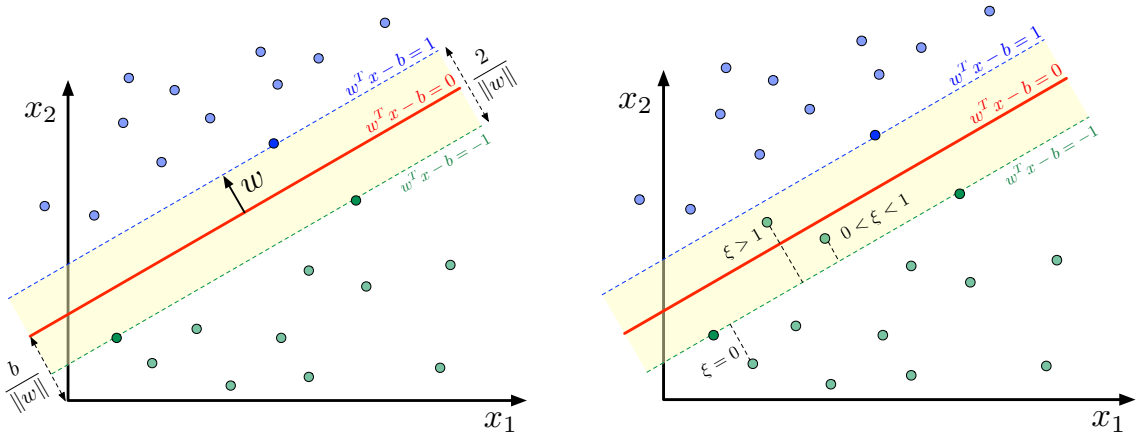

Figure 41.1: Illustration of a maximum margin classifier for a linear support vector machine in the separable case (left) and with the soft-margin and slack variables ξ (right).

Maximum-margin classifiers 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 41.1 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, $\hat{y}(x) = \operatorname{sgn}(w^T x - b)$.

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*.

Soft margins and slack variables Often the data will not be linearly separable and thus the hard-margin optimization problem described above has no solution. Such a non-separable case can be accommodated with the use of so-called *slack variables*. One slack variable $\xi_i > 0$ is introduced for each data point. As shown in Figure 41.1 (right), if a data point x_i is outside the margin and correctly classified as in the separable case, then $\xi_i = 0$. If it within the margin but still correctly classified, then $0 < \xi_i < 1$. If it is misclassified, then $\xi_i > 1$. The new constrained optimization problem is to minimize

$$\sum_i \xi_i + \lambda \|w\|^2 \tag{41.32}$$

subject to the constraints $y_i(w^T x_i - b) \geq 1 - \xi_i$ and $\xi_i > 0$ for $i = 1, \ldots, n$. Effectively, this corresponds to minimizing $\sum_i \max(0, 1 - y_i(w^T x_i - b)) + \lambda \|w\|^2$, where the first term is called the *hinge loss*.

The dual problem In the language of convex optimization, linear programming, and quadratic programming the optimization problems stated above for the hard and soft maximum-margin classifiers define the *primal* objective to be minimized. There is a *Lagrange dual* problem where the solution corresponds to maximization with $\min_\phi L(\phi) = \sup_{\alpha>0} \bar{L}(\phi, \alpha)$, where α are Lagrange multipliers. While it is not apparent in the primal formulation of the constrained optimization problems, the solution to the dual problem takes on the form

$$\hat{y}(x) = \text{sgn}(\sum_i y_i \alpha_i x_i^T x - b) \,, \tag{41.33}$$

where the $\alpha_i > 0$ for the support vectors and otherwise zero. This reformulation is plausible since the orientation of the maximum-margin hyperplanes defined by w is completely specified by the support vectors. The crucial observation for the kernel trick described below is that the final model is defined in terms of coefficients and the inner product $\langle x_i, x\rangle_{\mathcal{X}} = x_i^T x$.

The kernel trick One approach to obtain non-linear decision boundaries is introduce a non-linear mapping $\varphi\colon \mathcal{X} \to \mathcal{V}$ that embeds the data in some higher-dimensional space $\mathcal{V}$. One can then use the linear SVM described above in the space $\mathcal{V}$. This is not unlike the many neural network models where the first layers use a non-linear activation function while the last layer is linear.

The *kernel trick* avoids the explicit mapping φ that is needed to get linear learning algorithms to learn a nonlinear function or decision boundary. This is possible because the solution to the dual problem in Eq. 41.33 is represented in terms of inner products $\langle \varphi(x_i), \varphi(x)\rangle_{\mathcal{V}} = \varphi(x_i)^T \varphi(x)$ corresponding to the candidate point x and the support vectors x_i. Thus all that is needed is a way to evaluate the inner products. A *kernel* or a kernel function $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a function that can be expressed as an inner product in another space $\mathcal{V}$ via $k(x, x') = \langle \varphi(x), \varphi(x')\rangle_{\mathcal{V}}$. Critically, one does not need to ever evaluate the map $\varphi(x')$, which is implicitly defined by the kernel. In some cases the kernel is straight forward to evaluate, even though the target space $\mathcal{V}$ implicitly defined by the kernel would be infinite dimensional.

The choice of the kernel is roughly the analog of the architecture of a neural network. It dictates the types of non-linearities the SVM can implement in the input space. It also provides an opportunity to encode prior knowledge from physics such as symmetries, length scales, *etc.* There are also connections between very wide neural networks with random weights and biases and the kernel of a Gaussian process [187–189], which, when applied to deep networks, lead to the concept of neural tangent kernel [190]. While it is beyond the scope of this chapter, the interested reader may also be curious to understand the connections as formalized in the language of Reproducing kernel Hilbert spaces (RKHS) [191].

Support vector regression There are a few flavors of SVR. In Vapnik's ϵ-SVR formulation is similar to the picture for classification, but the boundary hyperplane serves as the prediction and the width of the margin for which there is no penalty is ϵ. Again, one can introduce a non-linear map $\phi(x)$ explicitly or implicitly via the corresponding kernel $k(x, x')$. One can view the predictive model as $\hat{y}(x) = w^T \phi(x) - b$ and the goal of the optimization problem is to minimize

$$\|w\|^2 + C \sum_i (\xi_i + \xi_i^*) \tag{41.34}$$

where ξ_i and ξ_i^* are slack variables associated to the model over or under-estimating $y_i \in \mathbb{R}$. The constrained optimization is subject to $w^T x_i - b - y_i \leq \epsilon + \xi_i$, $y_i - w^T x_i + b \leq \epsilon + \xi_i^*$, $\xi_i, \xi_i^* \geq 0$ for $i = 1 \ldots, n$. The solution to the dual problem leads to a kernelized representation of the solution

$$\hat{y}(x) = \sum_i (\alpha^* - \alpha) K(x, x_i) - b \,, \tag{41.35}$$

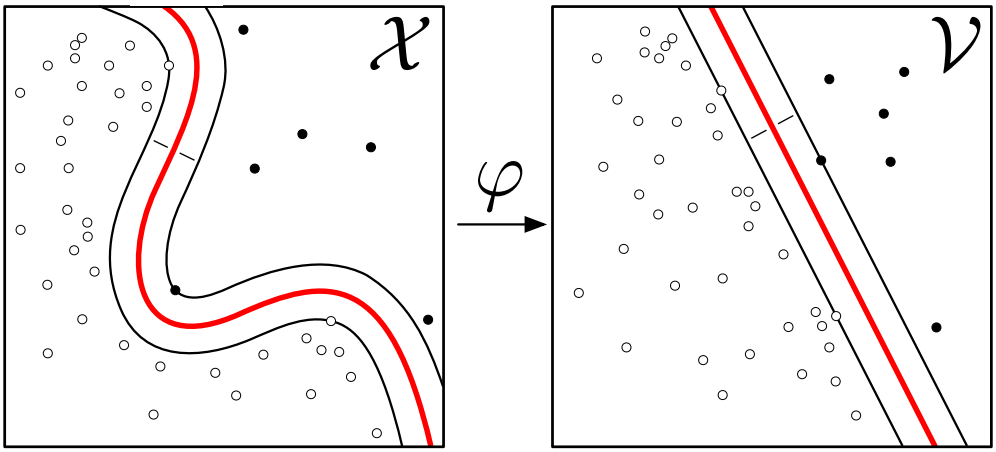

Figure 41.2: Figure depicting the nonlinear SVM decision boundary in $\mathcal{X}$ (left) and the corresponding linear maximum margin classifier in $\mathcal{V}$ (right). Figure adapted from Kernel_Macine.svg Wikimedia Commons (CC BY-SA 4.0) by Cranmer.

where the α_i, α_i^* are the parameters of the dual problem. In libraries like `LibSVM` the output of the algorithm is just $\alpha_i^* - \alpha_i$ for $i = 1 \ldots, n$.

Kernel ridge regression Kernel Ridge Regression (KRR) uses the same model as in SVR; however, it uses a different loss function. KRR uses the squared error loss while support vector regression uses the ϵ-insensitive loss function (*i.e.* no penalty is encured if the prediction is within ϵ). Both are combined with l2 regularization (the $\|w\|^2$ term). While SVR leads to a sparse solution that only depends on the support vectors, it can be slow to solve the constrained convex optimization problem. In contrast, KRR can be done in closed-form and is typically faster to train for medium-sized datasets. The consequence is that the learned model is non-sparse and thus slower than SVR to evaluate at prediction-time.

Gaussian Process Regression (krigging) Another variation on kernel-based regression is Gaussian process regression (GPR), also referred to as *krigging*. Again the model is based on the kernel trick. While KRR aims to minimize the mean-squared error loss $(y_i - \hat{y}(x_i))^2$ along with an l2 regularization term proportional to $\|w\|^2$, GPR elevates these terms to a probabilistic interpretation where the first is proportional to a log-likelihood for Gaussian $N(y_i|\hat{y}(x_i)), \sigma)$ likelihood and the second corresponds to the log of a Gaussian prior on w_i. This is related to the discussion on Tikhonov regularization found in Sec. 41.3.1.2. GPR goes further and uses a mean function $m(x)$ and kernel $k(x, x')$ to define a probabilistic model for $y(x)$ and $y(x')$ that includes correlations. For any finite number of points $\{x_j\}_{j=1,\ldots,m}$, the Gaussian process defines the joint distribution for the values that $f(x_i)$ might take. Typically, one specifies the prior mean $m(x)$ and prior covariance function $k(x, x')$ and then conditions on the training dataset $\{x_i, y_i\}_{i=1,\ldots,n}$. The posterior can then be evaluated for an independent set of points $\{x_j\}_{j=1,\ldots,m}$. The posterior mean is typically used as the prediction equivalent to SVR and KRR, but GPR also gives a meaningful notion of uncertainty in the predictions, and can be viewed as a realization of kernel Bayesian regression [192].

One advantage of GPR 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 SVR and KRR typically

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

Rasmussen and Williams provides an excellent review of Gaussian processes [193]. Numerically, Gaussian Process libraries are confronted with computing the log determinant of the covariance kernel, which naively scales like $\mathcal{O}(n^3)$ in computational complexity. However, recently there have been a number of numerical advances that make these methods fast and scalable to large datasets [194, 195]. While Gaussian Processes are used more widely in cosmology, they are not widely used in particle physics or astro-particle physics. 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 [196–198]. Gaussian Processes are also extensively used in Bayesian Optimization (Section 41.3.3).

41.5.2 *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.* $x_4 < 0.7$) [199]. The model is piecewise constant and assigns the value $b_j \in \mathcal{Y}$ to the j-th terminal region $\mathcal{X}^j$. The model can be written

$$\hat{y}(x) = f_\phi(x) = \sum_j b_j \mathbf{1}(x \in \mathcal{X}^j) \,. \tag{41.36}$$

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 that is composed of a mix of continuous and discrete features. Tools such as `XGBoost` [200] are widely used in data science competitions such as Kaggle and the boosted decision trees (BDTs) implemented in `StatPatternRecognition` [201] and `TMVA` [77] 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, one considers a sequences of pruned trees where splits are removed based on some heuristic. One finds the tree in this sequence of pruned trees that minimizes the generalization error on the validation set. Alternatively, in tools such as *XGBoost* there is an explicit regularization term included in the loss function (see Eq. 41.43).

Ensemble methods The idea of ensemble methods is to combine multiple models into a more performant one by exploiting the bias-variance tradeoff [202]. 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 averagaing 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

$$\hat{y}(x) = \sum_{t=1}^{T} \beta_t \hat{y}_t(x) \ , \tag{41.37}$$

where β_t expand the parameters of the model ϕ.

Bagging The idea behind bagging (bootstrap aggregating) 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 [203]. 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

$$\mathrm{err}_t = \frac{\sum_i w_i^{(t)} \mathbf{1}[y_i \neq \hat{y}_t(x_i)]}{\sum_i w_i^{(t)}} \ . \tag{41.38}$$

Based on this weighted error, the coefficient β_t of the component $\hat{y}_t(x)$ in Eq. 41.37 is given by

$$\beta_t = \log\left(\frac{1-\text{err}_t}{\text{err}_t}\right) . \tag{41.39}$$

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. [77] 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* [204]. In this setup, the model is purely additive as in the case of random forests, so the model is Eq. 41.37 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. 41.36. 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. 41.37. 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:

$$\text{obj}^{(t)} = \sum_{i=1}^{n}[\mathcal{L}(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \Omega(f_t) + \text{constant} , \tag{41.40}$$

where

$$g_i = \partial_{\hat{y}_i^{(t-1)}}\mathcal{L}(y_i, \hat{y}_i^{(t-1)}) \tag{41.41}$$

and

$$h_i = \partial^2_{\hat{y}_i^{(t-1)}}\mathcal{L}(y_i, \hat{y}_i^{(t-1)}) \tag{41.42}$$

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

$$\Omega(f) = \gamma J + \frac{1}{2}\lambda\sum_{j=1}^{J} b_j^2 , \tag{41.43}$$

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 j-th leaf. The heuristic used in `XGBoost` for splitting a node is

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma . \tag{41.44}$$

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.

41.5.3 *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. 41.6.1. Similarly, other important aspects to effectively training neural network models such as parameter initialization, early stopping, *etc.* are discussed in Sec. 41.6.

The vanishing and exploding gradient problem is a common challenge for gradient-based optimization of neural networks and is described in Sec. 41.6.5. That problem is referred to repeatedly in this section because it has motivated the development of numerous architectural components described below.

41.5.3.1 *Feed-forward multi-layer perceptron*

One of the core components in neural networks is the fully-connected, feedforward network or*multi-layer perceptron* (MLP), which is composed of L *layers*: $f = f^{(L)} \circ \cdots \circ f^{(1)}$. The l^{th} 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 responsible for producing an individual output in d_l-dimesional 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:

$$f^{(l)}(u) = \sigma^{(l)}(W^{(l)}u + b^{(l)}) \,, \tag{41.45}$$

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, $\phi = (W^{(1)}, \ldots, W^{(L)}, b^{(1)}, \ldots, b^{(L)})$.

41.5.3.2 *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 *logistic* and *hyperbolic tangent* functions. 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. 41.6.1). 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. 41.6.7, 41.6.8 and 41.6.9. 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 *Rectified Linear Unit* (ReLU):

$$\sigma(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{41.46}$$

which computational cost is small and provides the gradient does not vanish for $x \in (0, +\inf)$ [205, 206]. An alternative to preserve a non-zero gradient in negative input values are called *Leaky ReLU* and modifies the output to $0.01x$ for $x \in (-inf, 0)$ [207]. 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 [208].

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. Recently, in the area of

neural scene representation, sinusoidal activation functions have been found to be surprisingly effective [209].

41.5.3.3 *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 i-th component is given by:

$$v_i = \frac{\exp(u_i)}{\sum_{j=1}^{n} \exp(u_j)} \,. \tag{41.47}$$

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. 41.5.3.11).

41.5.3.4 *The rise of 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 non-linear activation function σ not a polynomial and the width d_1 is large enough [210]. However, it is often more efficient (in terms of the number of parameters) to increase the *depth* of the network L [211].

Training a deep network (*i.e.* $L > 2$) that generalizes well can be more 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 [212]), development of ReLU, research progress in initialization and optimization algorithms for model parameters, and regularization techniques like *dropout* [13] all played an important role in the rise of *deep learning* [2, 213]. 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.

41.5.3.5 *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 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

$$v_c(j) = (W * u_c)(j) = \sum_{c'} \sum_{i} W_{c,c'}(i) u_{c'}(\text{“}j - i\text{”}) \,, \tag{41.48}$$

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 Figure 41.3. Note that the the number of channels in the output v does not need to

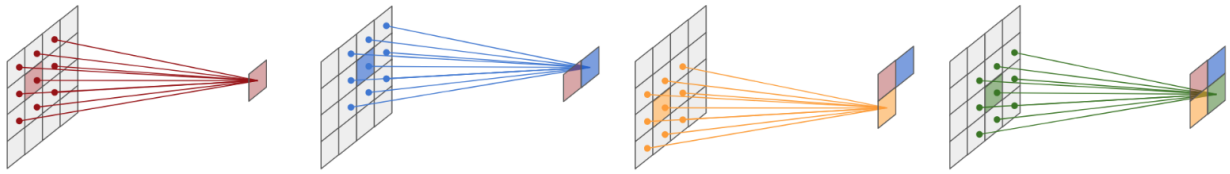

Figure 41.3: A pictorial description of a kernel convolution over four input pixels. It takes a product of the weight matrix (kernel) and the local input matrix centered at a target pixel. The operation is repeated over the input image using the same kernel. The size of the output image depends on the size of the kernel, stride, and padding. In this figure, the kernel size of 3, stride of 1, and padding of 0 is used.

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.

The discussion above makes it clear that CNNs will have fewer parameters than the corresponding fully connected MLP, which can alleviate the optimization challenge. In addition, the convolutional structure allows for data to be reused effectively as patterns in one part of an image in the training dataset effectively contribute to learning that pattern anywhere in the image. Larger kernel sizes allow for the filters to learn more complicated patterns, but at the expense of having more parameters and needing more data to train (the extreme case being a kernel size that is the the size of the entire image, which would be equivalent to a fully connected MLP). In practice, kernel sizes of 3 are popular, 5 is sometimes used, and larger kernels are rarely used.

A kernel convolution involves three hyperparameters, the size of kernel (typically an odd number so that the filter has an unambiguous center), *stride*, and *padding*. The stride is the number of pixels between each target pixel (*i.e.* the center of the filter). The stride size of 1 implies the target pixels are adjacent, and a stride of 2 implies skipping 1 pixel in between (along each spatial axis). Padding is an operation to expand the input image by a specified number of pixels (i.e. "padding size") for when the target pixel is near the edge and the filter would extend beyond the input image. In Figure 41.3, the input image is 4×4 pixels, the kernel is of size 3, and no (zero) padding is applied. The result is a smaller image. Alternatively, With the padding of size 1, the output would retain the 4×4 shape of the input image.

A kernel size of 1 is also frequently used and is referred to as a *1×1 convolution*. While it cannot capture a geometrical features, it can perform linear operations on the input features, including increasing or decreasing the number of features. When combined with a non-linearity and stacked repeatedly this can be seen as a small MLP attached to an individual pixel, and for this reason the 1×1 convolution is sometimes referred to as a *network-in-network* [214]. This is often used to

extract more powerful features to be used by the latter layers, and also to compress features when the next layers may be computationally demanding such as a block of many convolution layers with a large kernel size [215, 216].

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 4×4 kernel. In addition, a typical CNN architecture uses pooling (described below), which effectively down-samples 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 [215].

41.5.3.6 *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 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. 41.5.3.14). The pooling operation gives rise to robustness of the model to small, local deformations in the input, a property called *geometric stability* [217–219]. 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' even independent of where the dog is in the image).

On the practical side, it is also beneficial to reduce the dimensionality of the input to a smaller hidden state representation. This can be done either a convolution operation with the stride size larger than 1, or employing a *pooling*.

A typical CNN for extracting a 1-dimensional array of features is designed with repeating blocks of convolution layers and pooling operations [220]. Figure 41.4 shows an example evolution of a data tensor through the succession of a convolution and pooling operations in order 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 an 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 down-sampled feature maps. There may be more than one convolution layer within each spatial resolution (*i.e.* in between pooling operations). Following these design principles, CNN encoders typically become *deep*, consisting of dozens or sometimes hundreds of convolution layers, and face challenges of vanishing gradients problem (see Sec. 41.6.5). A standard practice to mitigate this issue is explicitly *normalize* the input tensor input at each convolution layer using algorithms like *Batch Normalization*. This will be discussed in Sec. 41.6.9.

41.5.3.7 *CNN architectures for image analysis*

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

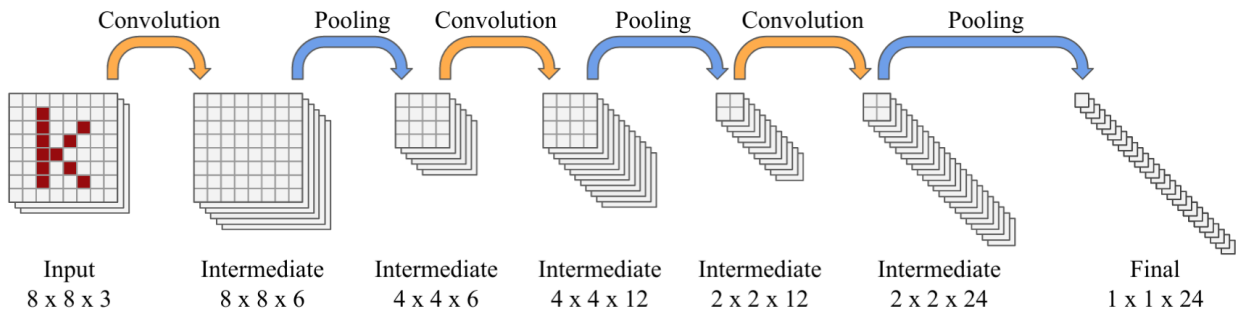

Figure 41.4: An example CNN architecture to extract a 1-dimensional array of features from an image via succession of convolution layers and pooling operations. The (square) kernel, stride, and padding size of a convolution operation are 3, 1, and 1 in respective order. The pooling operation uses a square 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.

- *Image classification or regression* requires a prediction of single value for the whole image (*i.e.* a category or target value)
- *Object detection* produces a list location information for arbitrary number of objects in the input image
- *Semantic segmentation* outputs an image of the same spatial dimension as the input, in which every pixel is segmented by a predicted value for a target task (classification or regression).

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 [221]. R-CNN is 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, 116, 222].

R-CNN consists of multiple CNNs. The first is a feature extractor which produces a spatially compressed feature tensor. The second CNN applies 1×1 convolution to predict two information: an *object score* which indicates whether there is an object in this (spatially compressed) pixel or not, 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 *Region Proposal Network* (RPN), and the bounding box is called *Region of Interest* (ROI). The the size of a bounding box is not directly solved by a regression. Instead, the model requires a set of pre-defined *anchor* boxes, which sizes and aspect ratios are hyperparameters, and predict the object score, location, and size (as a multiplicative factor to the defined anchor box) for each anchor box. This allows RPN to detect multiple objects with different aspect ratios even if they are within the same receptive field in the original input image. 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. Given a list of bound-

ing boxes with confidence scores, NMS takes the box with the highest score, and computes the Intersection-over-Union (IoU) of this box and all the rest of boxes. Boxes with the IoU higher than a threshold (hyperparameter) are eliminated as they are likely to represent the same object. Then NMS repeats the same operation for the box with the second highest score among the remaining boxes. This is repeated till the list is exhausted.

U-Net is one of the of most successful models used for semantic segmentation [223]. As the output of U-Net is also an image, it gives some interpretability compared to models for an image-level classification or regression tasks. The model is used widely in HEP experiments in both 2D and 3D image data [112, 113, 118, 125, 127].

The architecture of U-Net consists of a CNN encoder and *decoder*. A decoder consists of convolution and *convolution-transpose* layer (also called strided transpose-convolution, or rarely deconvolution). The operation of a convolution transpose can be seen an opposite of a convolution: for every pixel in an input image, its value is multiplied to the kernel and copied to the output. $3x3$ kernel in transpose-convolution layer fills $3x3$ pixels in the output tensor. The concept of padding and the stride are also applied to the output tensor. For instance, the stride 2 transpose-convolution fills $3x3$ fields centered at every other pixel (*i.e.* stride 2) for every input pixel. The spatial size of an output tensor is therefore determined by the spatial size of an input tensor, stride, and padding. In the decoder of U-Net, convolution-transpose layers are used to up-sample spatially compressed feature tensors back to the original image resolution. The stride size of a convolution-transpose layer is almost always 2 in order to *slowly* transform the spatial size, and standard convolution layers (with stride 1) are placed in-between up-sampling operations. Because the output of the decoder has the same size as the input image, features for every pixel can be used for either a classification or a regression task at the pixel-level.

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 down-sampling operation (*e.g.* max-pooling) in the encoder is, however, a lossy process where spatial information is permanently lost. This prevents a simple encoder-decoder architecture to perform a semantic segmentation task at the level of precision required for science research and industrial use. Interesting observation made by the U-Net authors is a symmetry in the encoder-decoder architecture in which there are tensors of corresponding (*i.e.* the same) spatial size between two parts of the model. This observation inspired them to concatenate the intermediate tensors in the encoder block to the intermediate tensors of the corresponding size in the decoder block before convolution layers are applied in the decoder, which dramatically improved the performance of semantic segmentation. This is a type of a *skip connection* discussed below.

41.5.3.8 *Residual networks and skip connections*

The representation power 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. 41.6.5). One of powerful techniques to address this challenge is a *residual network* (ResNet), which is a modular architecture design that can be applied to neural network models [216]. 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 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 $1{\times}1$ convolutions to transform and match the dimension. The design is modular as it can be applied per a stack of convolution layers (e.g. U-ResNet introduced

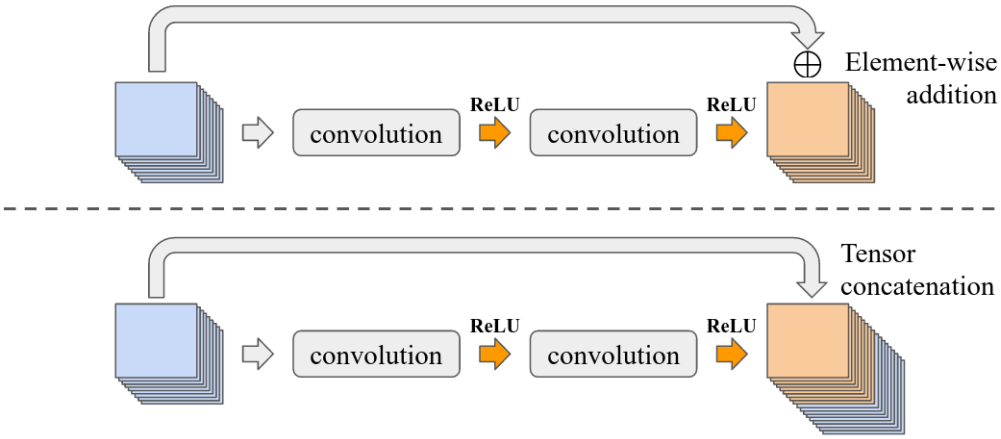

Figure 41.5: Two types of skip connections: the top is from ResNet where the input is element-wise added to the output tensor of a block of convolution layers while the bottom shows a concatenation of the input to the output tensor as employed in other models including U-Net and DenseNet.

for LArTPC detectors uses ResNet modules within a U-Net architecture [112, 113]) widely in the present applications.

The authors of ResNet successfully demonstrated an improved performance of some models at the depth exceeding 1000 layers where the non-residual counter part could not improve the accuracy beyond a few dozens of layers. This remarkable success was due to the *skip connections* in ResNet, namely the element-wise addition of the input to the output at each block of convolution layers as shown in Figure 41.5. Without the presence of the skip connections, the gradients from the loss function is forced to go through all convolution layers (i.e. the reverse direction of orange allows in Figure 41.5), which results in the gradient being altered at every convolution layer. The layers closer to the input receive altered gradients by the continuously updated weights of deeper layers, making conversion difficult and slow. The skip connections provides the path for the gradient to flow directly to the preceding layers. As a result, the gradient can flow more directly to preceding layers and allows simultaneous optimization across layers. It is possible, however, that some convolution layers in ResNet may learn an identity mapping, thus contributing no effect to solving a given task despite consuming computing power.

Another type of skip connections is a concatenation of feature maps, in which case the input and the output of a block of convolution layers do not necessarily have to have the same number of features. This is shown to be very powerful in number of popular model architectures including U-Net [223] and DenseNet [224]. In this case, the input features are re-used: the next block of hidden layers can learn an optimal way to combine the input and the output of the preceding block (i.e. it could learn a simple addition operation, which makes it identical to ResNet). In DenseNet, input to a convolution layer is concatenated with the output of all preceding convolution layers of the same spatial dimension, thus $x_l = f_l([x_1, x_2, \cdots, x_{l-1}])$ where x_l is the output of l-th layer f_l and $[x_1, \cdots, x_{l-1}]$ denotes a concatenation of the inputs. In U-Net, tensors from the encoder block are concatenated with the tensors in the decoder block where they have the same size in spatial dimensions. This was critical for achieving a high spatial resolution for *semantic segmentation*, a class of tasks in Computer Vision to classify every pixel in an image among the predefined set of types (semantics), because the geometrical features at each spatial resolution is otherwise lost via pooling operation in the encoder block [223].

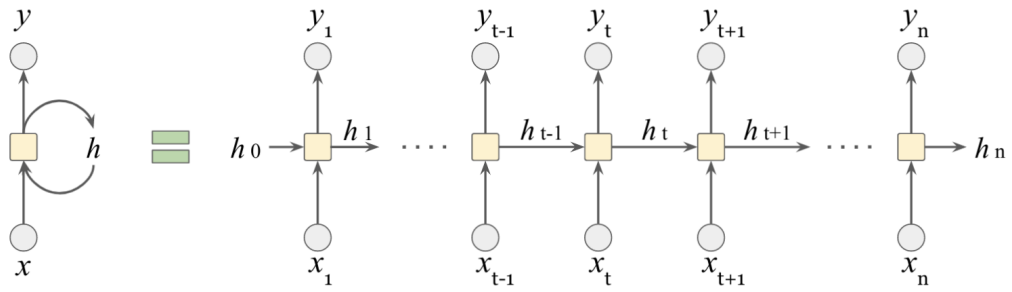

Figure 41.6: Pictorial description of a RNN (on the left) which takes an input and produces an output at every step with a hidden-to-hidden connection. The right diagram is unrolled over discrete steps. The yellow box represents a cell: a set of operations unique to each architecture.

41.5.3.9 *Recurrent neural networks*

Recurrent Neural Networks (RNNs) [225] 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:

$$h_t = g_h(h_{t-1}, x_t, \theta) \tag{41.49}$$

where h_t and θ denote the *hidden state* of the system and parameters of f, 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 41.6 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:

$$\begin{aligned} h_t &= g_h(Wx_t + Vh_{t-1} + b) \\ y_t &= g_o(Uh_t) \end{aligned} \tag{41.50}$$

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 clasification accuracy over the sequence.

Variations in RNN architectures result from the design of cells (described below) and flow of information across the cells. For instance, a bi-directional RNN (Figure 41.7 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 41.7 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.

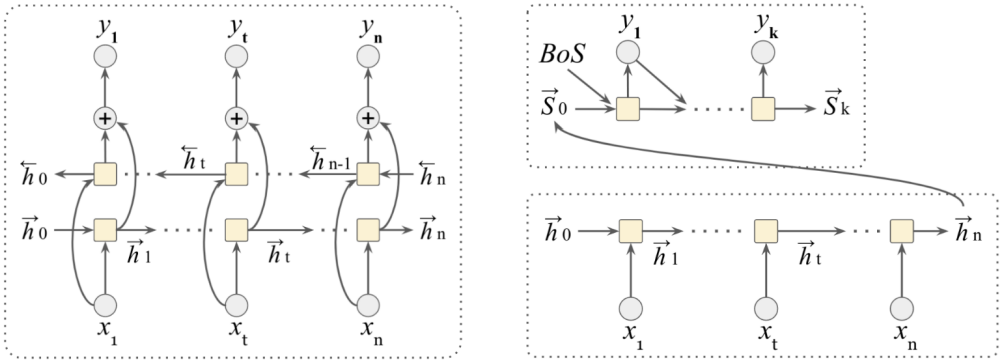

Figure 41.7: Bi-directional RNN (left) provides contexts in the preceding and subsequent parts of the input sequence. RNN encoder-decoder (right) can generate an output with a different sequence length from an input. Each cell in the decoder may take a previously generated element, starting from a special marker that signals the beginning of the sequence (BoS) and ending when the end of sequence is generated.

41.5.3.10 *LSTM and GRU*

An RNN applies the same functions g_h and g_o in Eq. 41.50 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 [226, 227]. 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. 41.50) can lead to gradients that can either explode or vanish (see Sec. 41.6.5). 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) [228] 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:

$$\begin{aligned} c_t &= f_t \bigodot c_{t-1} + i_t \bigodot \tilde{c}_t \\ h_t &= o_t \bigodot c_t \end{aligned} \qquad \text{where} \qquad \begin{aligned} f_t &= \sigma\left(W^f x_t + V^f h_{t-1} + b^f\right) \\ i_t &= \sigma\left(W^i x_t + V^i h_{t-1} + b^i\right) \\ o_t &= \sigma\left(W^o x_t + V^o h_{t-1} + b^o\right) \\ \tilde{c}_t &= \tanh\left(W^c x_t + V^c h_{t-1} + b^c\right) \end{aligned} \tag{41.51}$$

where σ and $\bigodot$ 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 41.8, one can see

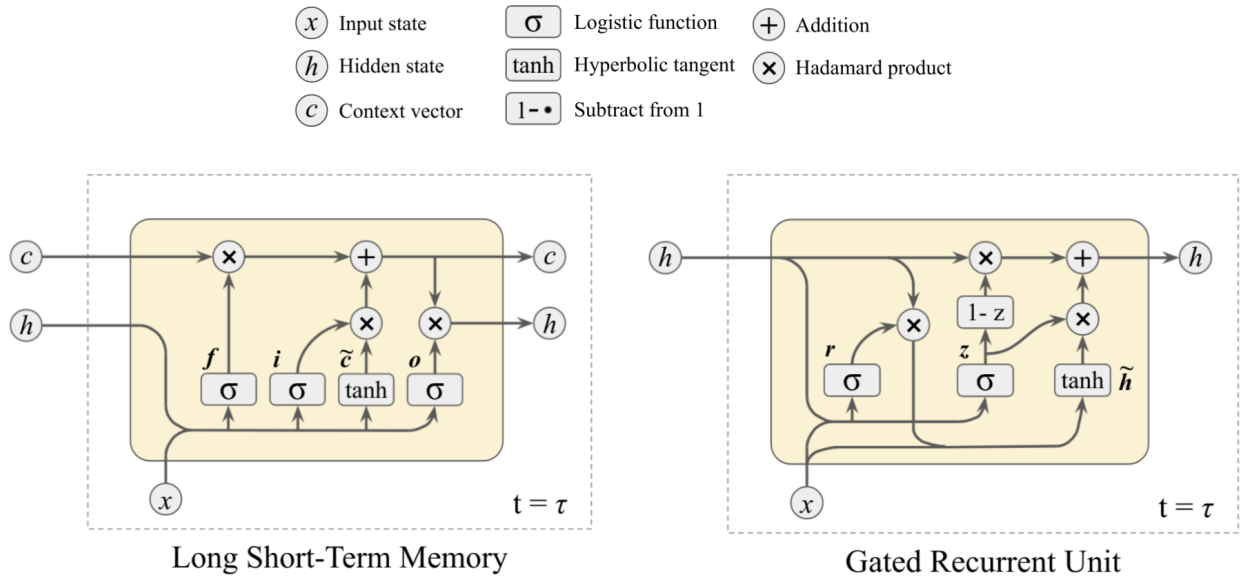

Figure 41.8: LSTM (left) and GRU (right) are both gated neural network designed to address a vanishing gradient problem for RNNs.

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. 41.5.3.8), which enabled training of CNNs with thousands of layers.

Another gated model to solve a vanishing gradient problem is the *Gated Recurrent Unit* (GRU) [182]. 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:

$$h_t = z_t \bigodot h_{t-1} + (1 - z_t)\tilde{h}_t \qquad \text{where} \qquad \begin{aligned} r_t &= \sigma\left(W^r x_t + V^r h_{t-1} + b^r\right) \\ z_t &= \sigma\left(W^z x_t + V^z h_{t-1} + b^z\right) \\ \tilde{h}_t &= \tanh\left(W^h x_t + V^h \left(r_t \bigodot h_{t-1}\right) + b^h\right) \end{aligned} \tag{41.52}$$

where r_t and z_t are referred to as *reset* and *update* gate. As one can see in Figure 41.6, 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 [153, 154, 229–231].

41.5.3.11 *Attention*

Many tasks encountered in machine learning can be divided into subtasks. For example, classifying each pixel of an image or element of a sequence, predicting a target for each node in a graph, *etc.*. For each subtask the model may need to draw upon information from the entire input, but some parts of the input will be more relevant than others. At some point, the model will need to

form a representation for the input as context for the subtask at hand. A fixed length vector c as a global context for this subtask is possible, but this approach scales poorly as the size of the input and number of subtasks grow. Either the size of the global context must grow or it will not have the capacity to represent all the relevant information, which would lead to a degredation in performance.

The idea behind *attention* is intuitive: one still forms a representation for the entire input, but different parts of the input are weighted differently according to the task at hand. By making the weights a learnable component, the network can learn to attend to the relevant parts of the input. A softmax function is a natural way to represent attention as it assigns a positive value to each component of the input and sums to one. Then for the i-th task, one can simply 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. Let α_{ij} be the weights assigned to the j-th input for the i-th task, such that $\sum_j \alpha_{ij} = 1$. We can satisfy this naturally if for each i the α_{ij} are computed from the logits β_{ij} using a softmax function that normalizes by summing over j. Putting these ingredients together, we have the *additive attention mechanism*

$$c_i = \sum_{j=1}^{n} \alpha_{ij} h_j \qquad \text{where} \qquad \alpha_{ij} = \text{softmax}\left(\beta_{ij}\right)_{\text{over } j} . \tag{41.53}$$

As in the case of a multi-class classifier, the logits β_{ij} can be computed from a network component with learnable parameters. For instance, in the case of a cell of an RNN encoder-decoder network (see Fig. 41.7) that is decoding element i with an incoming input state s_{i-1}, the logits for the attention mechanism could be computed as follows

$$\beta_{ij} = U \tanh\left(W s_{i-1} + \tilde{W} h_j + b_i\right) , \tag{41.54}$$

where U, W, $\tilde{W}$ are the weights and b is the bias term of the model. Fig. 41.9 from Ref. [232] illustrates the full attention mechanism. The idea was implemented by a model called *RNNSearch* which made a breakthrough in machine translation by combining a bi-directional RNN with an additive attention mechanism [233].

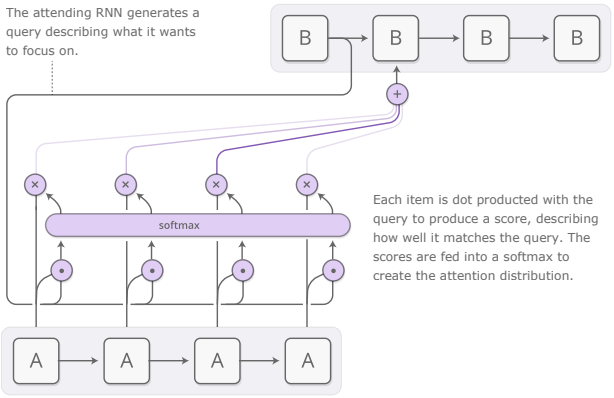

Figure 41.9: An illustration of the attention mechanism From Olah and Carter, "Attention and Augmented Recurrent Neural Networks". Lower boxes labeled `A` represent input elements in the sequence and upper boxes labeled `B` indicate output elements. The left-most line originating from the first `B` corresponds to the state s_{i-1} in the text.

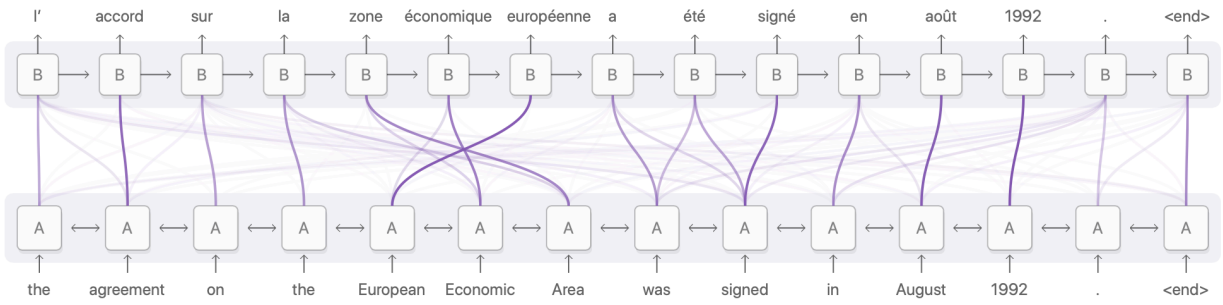

Figure 41.10: Visualization of the attention weights in a sequence-to-sequence problem taken from Olah & Carter, "Attention and Augmented Recurrent Neural Networks" The thickness of the lines is proportional to the attention weights α_{ij}..

We end this section by noting that the softmax α_{ij} can be used to visualize the influence of the j-th input element on the i-th output eleement, which improves interpretability of the model. An example of this from Ref. [232] is shown in Fig. 41.10.

41.5.3.12 *Scaled dot-product attention*

Shortly after the introduction of RNNSearch, the attention mechanism has been recognized as a powerful tool. One such variant is *scaled dot-product attention*, which is most widely recognized as the foundation of the *Transformer* architecture [234], which is described in more detail below.

In additive attention (Eq. 41.53) the hidden representations h_j were combined through a weighted average based on the coefficients α_{ij}, resulting in a task-specific context vector c_i. In the literature around scaled dot-product attention, multi-head attention, and transformers, the hidden states that will be combined are referred to as *values*, and they 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. 41.53 could be rewritten as $c = \text{softmax}(\beta)V$, where the softmax is normalized per row.

In scaled dot-product attention, the basic structure will be different, but instead of using an non-linear network component to compute the logits β as in Eq. 41.54, the logits will be computed by forming a dot product between a in incoming *query* and a *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) $K^T \in \mathbb{R}^{d \times m}$. When the dot product between a particular query vector $q_i \in \mathbb{R}^d$ and key vector $k_j \in \mathbb{R}^d$ is large, then the resulting logit β_{ij} and attention weight α_{ij} will be large. 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 scalled by a factor of $1/\sqrt{d}$. The resulting task-dependent context is $c_i = \text{softmax}_j(q_i \cdot k_j/\sqrt{d})v_j$, where softmax_j indicates that normalizing sum runs over the index j. A common, though sometimes confusing, notation is simply

$$c = \text{Attention}\,(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V\,, \tag{41.55}$$

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

41.5.3.13 *Transformer and multi-head attention*

The Transformer architecture is a powerful encoder-decoder model based on the scaled-dot product attention mechanism. It was originally designed for sequential data like RNNSearch, and subsequently used in other areas of research including computer vision. One advantage of scaled-dot product attention compared to the approach used in *RNNSearch* (Eq. 41.54) is that computing the logits β_{ij} (and thus the attention weights) does not involve any sequential processing. This allows the models to better leverage parallelism of the hardware to train much more expressive models than before. In place of the gated unites 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 41.5).

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

Models such as these have made breakthroughs in many areas of scientific and industrial research [235].

41.5.3.14 *Graph networks and geometric deep learning*

Graphs are a powerful data structure 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 be seen as a generalization of many common types of machine learning models such as recurrent networks, convolutional neural networks, *etc.* [219]. 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. [236]. 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 i-th 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 k-th 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 and three *aggregate* functions ρ:

$$\begin{array}{lll} \mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}) & \mathbf{e}'_i = \rho^{e\to v}(E'_i) & \text{where} \quad E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e} \\ \mathbf{v}'_i = \phi^v(\bar{\mathbf{e}}_i, \mathbf{v}_i, \mathbf{u}) & \bar{\mathbf{e}}' = \rho^{e\to u}(E') & \text{where} \quad E' = \cup_i E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e} \\ \mathbf{u}' = \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}) & \bar{\mathbf{v}}' = \rho^{v\to u}(V') & \text{where} \quad V' = \{\mathbf{v}'_i\}_{i=1:N^v} \end{array} \tag{41.56}$$

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 i-th 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 [237] (GNNs) are the class of neural networks that work on graph-structured data. A direct analog in Computer Vision and physics is the *point cloud* data type, which is an unordered set of points. 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 [238, 239], 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 [240], which could be considered as a separate graph operation, to introduce rotation and translation invariance and to capture topological features). Deep Sets [241] follows 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 [242] 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 Non-Local Neural Network [243] to efficiently propagate local feature information to points that may be far in the 3D cartesian coordinate. Message Passing Neural Network [244] (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 [117, 121, 132, 142, 156, 183, 245, 246], 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 often offer a natural way to organize the entities in the data and encode how those components should interact each other. This particular type of inductive bias is referred to as *relational inductive bias* in Ref. [237]. The structure of the graph is both an opportunity and a responsibility as one needs to define the graph structure including the edges to define the mode. 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, and Minimum Spanning Tree.

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. Alternatively, one can formulate clustering as an edge classification task. A comprehensive review on particle physics applications have been made available recently [247].

41.5.4 *Deep 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. 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 distribuiton $p(x, y)$ in a supervised setting). An enlightening discussion of these two approaches can be found in Ref. [10].

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 Auto-Encoders (VAEs) [248, 249] , Generative Adversarial Networks (GANs) [250, 251], and Normalizing Flows (NFs) [252–256], though there 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. 41.18) as described in Sec. 41.3.2.1. 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. 41.3.1.3 and the discussion around the reverse KL divergence below Eq. 41.16). 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 liklelihood is for almost all data is zero, and so even ELBO-based training will not work. The breakthrough idea introduced in Ref. [250] 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, but this may cause confusion as the relationship between z and the observed x is deterministic. Typically, one would reserve the term latent variable for situations where one must marginalize (integrate) over z to obtain the marginal likelihood as in $p(x) = \int p(x, z)dz$. In this integral would involve a delta function imposing the deterministic relationship $x = g(z, \theta)$. A more natural interpretation of the relationship between x and z is through the change of variables formula, which is the essence of normalizing flows.

In the case of VAEs, the 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 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 generted 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 [257, 258]. 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 [44, 46, 252, 259, 260]. 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 [255, 261–263], and to achieve the best performance on a new dataset one needs extensive hyperparameter explorations [264].

41.5.4.1 *Variational auto-encoders*

The auto-encoder was described in Sec. 41.3.2.2 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 auto-encoder is not a probabilistic model, but additional probabilitic structure can be added.

One approach is VAE mentioned above [248, 249]. By equipping the latent space with a prior distribution $p(z)$, the decoder of the auto-encoder $g(z, \theta)$ 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

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \ . \tag{41.57}$$

While the likleihood 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 uses optimization to optimize the ELBO with respect to its parameters ϕ.

$$\text{ELBO} = \mathbb{E}_{q(z)} \log p(x|z) - D_{KL}[q(z)||p(z)] \leq \log \mathbb{E}_{q(z)} \left[\frac{p(x,z)}{q(z)} \right] = \log p(x) \ , \tag{41.58}$$

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

$$D_{KL}[q(z)||p(z)] = \mathbb{E}_{q(z)}[\log q(z) - \log p(z)] \geq 0. \tag{41.59}$$

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 effeciently with a single forward pass of the neural network. Note the standard auto-enocder 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 auto-encoder, which only minimizes the reconstruction error, ELBO optimization of Eq. 41.58 has a trade-off 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 [265]. 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 [250], 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 auto-encoder 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 auto-encoder, PCA is not a probabilitic model, but probabilistic structure can be added. Probabilistic PCA [266] 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 non-linear encoder and decoder as in an auto-encoder. A Gaussian prior is a poor ansatz for the latent space distribution of data proceed by an auto-encoder. 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 [46] and in Probabilistic Auto-Encoder (PAE) [44], 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.

41.5.4.2 *Generative adversarial networks*

GANs [250] 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 likleihood $p(x)$) 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

$$\min_g \max_D \mathcal{L}_{\text{GAN}} = \min_g \max_D \{\mathbb{E}_{x\sim p(x)} \log D(x) + \mathbb{E}_{z\sim p(z)} \log[1 - D[g(z)]]\}. \tag{41.60}$$

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 distriminator.

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. 41.60 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 [257, 258]. One of the core issues is that the distribution generated by the GAN is restricted to a manifold and the KL divergence isn't well defined in this case (because p is not absolutely continuous with respect to q). To address these issues Wasserstein GANs train on

$$\min_g \max_D \mathcal{L}_{WGAN} = \min_g \max_D \{\mathbb{E}_{x\sim p(x)} D(x) - \mathbb{E}_{z\sim p(z)} D[g(z)]\}. \tag{41.61}$$

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. 41.61 can be interpreted as the dual form of the 1-Wasserstein distance between the true and generated distribution [267]. Wasserstein distances are a measure of dissimilarity between two distributions used in the context of Optimal Transport, mathematical theory of how to optimally transport one distribution to another. 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. 41.61 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 [268].

Because of the discriminative nature of the dissimilarity measure defined in data space, GANs often generate more realistic samples than VAE or normalizing flows in high dimensions such as natural images. However, GANs do not provide an encoder from data to latent space, do not provide a tractable likelihood $p(x)$.

41.5.4.3 *Normalizing flows, autoregressive models, and score based models*

Normalizing Flows (NF) provide a powerful framework for density estimation and sampling [252–256, 269]. 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 modelled as a random number with a simple base distribution $p_Z(z)$, which is typically chosen to be a standard normal (Gaussian) distribution. The probability density of the model be evaluated using the change of variables formula:

$$p_X(x) = p_Z(f(x)) \left|\det\left(\frac{\partial f(x)}{\partial x}\right)\right| = p_Z(f(x)) \prod_{l=1}^{n} \left|\det\left(\frac{\partial f_l(x)}{\partial x}\right)\right|, \tag{41.62}$$

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 embeded in the data space [44, 46, 252, 259, 260]). As such, they do not have the problems of GANs and VAEs they can be trained via maximum likelihood (Eq. 41.18) as described in Sec. 41.3.2.1.

There are several popular architectures of NFs. A method used by NICE, RealNVP and Glow [253–255] 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

$$x_2 = \exp(s(z_1)) \odot z_2 + m(z_1), \tag{41.63}$$

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 [270].

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 [271] and other variants.

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

$$p(x) = \prod_{i=1}^{n} p(x_i|x_1, x_2, \ldots, x_{i-1}) \; . \tag{41.64}$$

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 [272] and PixelCNN [273]) 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) [269].

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 [45]. Along each slice direction one obtains the 1d marginal distribution that can be mapped to a Normal distribution using a cumulative distribution function method. In 1d the Jacobian and the inverse transformation are tractable. The projection directions are chosen to maximize the sliced 1d Wasserstein distance between the data and the distribution generated by the samples, or between the inverse flow from the data to the latent space and the target distribution (typically chosen as the Normal distribution). Because this training is less sensitive to small variance directions than maximum likelihood training it achieves better results on anomaly detection tasks [45].

One can reduce the architectural restrictions imposed by normalizing flows or autoregressive models by modeling the "score" $\nabla_x \log p(x)$ instead of $p(x)$. Note this usage of the term "score" is non-standard and goes back to Ref. [274]; the standard use of the term score is theta of Eq. 41.26. Score-based training avoids 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 resulting generative models, called score-based generative models [275] or diffusion probabilistic models [276], have several advantages over existing model families. They achieve 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 [276]. The main advantage is that the distribution $p(x)$ can be specified solely by its gradient, which can subsequently be sampled from using Langevin dynamics. This is similar to the gradient based Monte Carlo Markov Chain methods (such as Langevin or Hamiltonian Monte Carlo) that sample from Bayesian posteriors without directly estimating the normalizing constant. This in turn enables more flexible model architectures than what can be used in normalizing flows or autoregressive models.

We end by noting that normalizing flows, autoregressive models, and other deep generative models that provide a tractable likelihood are incredibly 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. 41.3.5, 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 $p(\theta|x)$ [56, 57]. These techniques are being actively explored and applied to a number of scientific problems.

41.6 Learning algorithms

41.6.1 *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. 41.6.7 for neural networks), and perform an iterative update:

$$\theta_t = \theta_{t-1} - \lambda \nabla_\theta \mathcal{L}(x, \theta), \tag{41.65}$$

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 $\delta(\nabla_\theta \mathcal{L}(x, \theta))$:

$$\delta\left(\nabla_\theta \mathcal{L}(x, \theta)\right) \approx \delta\theta \cdot \nabla_\theta \mathcal{L}(x, \theta) = -\lambda |\nabla_\theta \mathcal{L}(x, \theta)|^2 \tag{41.66}$$

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* . 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. 41.6.7.

41.6.2 *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 $N < n$,

$$\nabla_\theta \mathbb{E}_{\hat{p}(x)} \mathcal{L} \approx \frac{1}{N} \sum_i^N \nabla_\theta \mathcal{L}_i, \tag{41.67}$$

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. 41.6.6 and Fig. 41.11).

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 [277]. 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 Adaptivee Rate Scaling (LARS).

41.6.3 *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.

One way to make the learning rate dependent on θ_i is to consider the gradient norm squared $(\nabla_{\theta_i}\mathcal{L})^2$. RMSprop learns its running average and then reduces the learning rate in directions with a large average gradient norm squared, thereby reducing the oscillations along that direction. ADAM (Adaptive Moment Estimation) combines the momentum and gradient norm ideas, computing running averages of both the gradient and the gradient norm squared, each with its own forgetting factor [278].

41.6.4 *Automatic differentiation and back propagation*

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:

$$\nabla_{\theta_i}\mathcal{L} = \frac{\partial\mathcal{L}(f(x,\theta))}{\partial\theta_i} = \frac{\partial\mathcal{L}}{\partial x_n}\cdot\frac{\partial x_n}{\partial x_{n-1}}\cdots\frac{\partial x_i}{\partial\theta_i} \tag{41.68}$$

where x_n denotes the output of n-th composite function f_n. In order to compute $\nabla_{\theta_i}\mathcal{L}$ for f_i, it needs computation of a gradient at all preceeding (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 *back propagation* [225]. An example of f that satisfies conditions to apply back propagation 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, division) and functions (e.g. log, exp, sin, 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 back propagation 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. 41.68. 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 $k << \ell$ while the reverse mode takes an advantage if $\ell << k$. For instance, in the case of an image classification where $(k,\ell) = (\text{pixel count}, 1)$, the reverse AD is more efficient.

41.6.5 *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. 41.68, the gradient of i-th 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. 41.50, we can write the back-propagating gradient:

$$\frac{\partial h_t}{\partial h_{t-1}} = \text{diagonal}\left(f'\left(Wx_t + Vh_{t-1} + b1\right)\right) W \tag{41.69}$$

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

$$\frac{\partial \mathcal{L}_i}{\partial h_j} = \frac{\partial \mathcal{L}_i}{\partial h_i} V^{i-j} \prod_{j<t\leq i} \text{diagonal}\left(f'\left(Wx_t + Vh_{t-1} + b1\right)\right) \tag{41.70}$$

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 [226, 227] 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

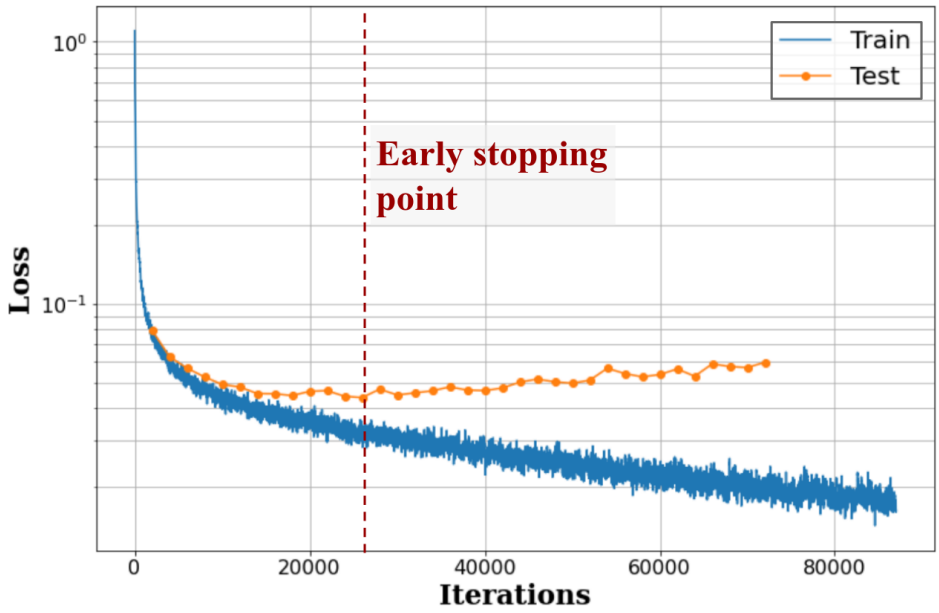

Figure 41.11: An example instance of overfitting. The training loss (vertical axis) shown in blue decreasing over iterations (horizontal axis) while the loss values evaluated on test samples shown in orange start to increase at around 26,000 iterations as indicated by the vertical line.

the back propagation:

$$\delta = \frac{\delta_{\max}}{\|\delta\|}\delta \text{if} \|\delta\| > \delta_{\max}. \tag{41.71}$$

This is called *gradient clipping* [227].

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) [279]. These factors are discussed in the following.

41.6.6 *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. 41.11. 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 [280]; however, the most straight forward approaches use a held-out validation dataset to monitor the generalization performance [281].

41.6.7 *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) [282]. 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 [220]. The *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 [283]. 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 [208].

41.6.8 *Input normalization*

Input data to a neural network is often pre-processed for the same goals discussed previously: values are shift 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 [279].

41.6.9 *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* [284] 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*, which shifts and scales the input to a hidden layer:

$$\tilde{u}^{(l)} = \gamma \frac{u^{(l)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \tag{41.72}$$

where $u^{(l)}$ and $\tilde{u}^{(l)}$ refer to the raw and normalized input to the l-th 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 [285], Instance Normalization groups along the spatial dimension but not along the batch nor channel [286], and Group Normalization is similar to Layer Normalization but forms multiple sub-groups of channels [287]. These variants does not apply normalization across samples within a batch, and thus agnostic to the batch size.

41.6.10 *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. Two tasks and data sets for each task may be different: the idea is that fundamental features may be reusable across different data and tasks. Transfer learning typically takes two steps: the first is to alter the model or data if necessary, then train (*fine-tuning*) the model. The first step is required, for example, when solving a different task that requires a different architecture (e.g. regression v.s. 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 [288–291]: 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 generic AI as well as interdisciplinary sharing of models across research fields.

For sequential models, transfer learning had been challenging before the appearance of Transformer. While Transformer was initially introduced for machine translation, Generative Pre-trained Transformer (GPT-1) showed that the model can be generalized to multiple NLP tasks by achieving the state-of-the-art in several seemingly different tasks including a sentence classification, semantic similarity, question answering and commonsense reasoning [292].

41.6.11 *Zero, one, and a few shot learning*

Fine-tuning (and thus transfer learning) may not be necessary if model is well generalized: humans can picture an imaginary animal in mind just from descriptions, or perform a task that he or she has never done before. One-shot learning is an extreme case of transfer learning where only one example is presented at the fine-tuning stage [293]. A similar variant is called a few-shot learning (fine-tuning using a few examples). This is possible only if the model has already learned the solution during the pre-training and the example is used to map the solution to the task. Zero-shot [294–296] learning is even more extreme where the model is given a new task without any example. This is only possible if the task is already learned during the pre-training in an implicit or unsupervised manner since zero-shot training implies that the model was never trained for the task.

41.7 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. [297] 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 mismodelling 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 mismodelling will influence results when applied to real data (*i.e.* $p_{\text{prod}}(x, y)$).

One of the primary approaches to incorporating this type uncertainty is to introduce nuisance parameters ν corresponding to the uncertain inputs to the simulation and to parameterize various types of perturbations (*e.g.* corrections to efficiencies, energy scales, *etc.*) in 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 performance 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 modelling of the data distribution. We return to this point in Sections 41.7.5 and 41.7.6.

41.7.1 *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 $\epsilon_y = P(f(x) > f_{\text{cut}}|y) = \int H(f(x) - f_{\text{cut}})p(x|y)dx$, 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 or background, electron or jet, *etc.*). 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. [298].

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 $\epsilon_y(\nu) = P(f(x) > f_{\text{cut}}|y,\nu) = \int H(f(x) - f_{\text{cut}})p(x|y,\nu)dx$. 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.

41.7.2 *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-modelling 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. [299] introduced an adversarial approach (similar to what is used in the generative adversarial network of Sec. 41.5.4.2) 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, squared-error, *etc.*) 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

$$\hat{\phi}_f, \hat{\phi}_r = \arg\min_{\phi_f} \arg\max_{\phi_r} E_\lambda(\phi_f, \phi_r) \,, \tag{41.73}$$

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

$$E_\lambda(\phi_f, \phi_r) = \mathcal{L}_f(\phi_f) - \lambda \mathcal{L}_r(\phi_f, \phi_r) \,. \tag{41.74}$$

The constant λ is a hyperparameter, since generially 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

$$\mathcal{L}_r(\phi_f, \phi_r) = \mathbb{E}_{p(x,y,\nu)}[-\log q_{\phi_r}(\nu|f_{\phi_f}(x))] \tag{41.75}$$

is an example of conditional density estimation (see Sec. 41.3.2.1), 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 [300–302], while here ν parameterizes

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. 41.73 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. [303] 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. [299] 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. [304] and [303] 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.

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 [304]. Widely used alternative approaches to decorrelation include uboost [305] and DDT [306]. Other examples of the domain adaptation and decorrelation use cases from the Living Review include [299, 303–318].

41.7.3 *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. 41.74 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. [58] in the context of simulation-based inference (see Sec. 41.7.7) and in Ref. [319] for new physics searches. It has also been applied to simulation-based inference for effective field theory parameters in Ref. [19] and Ref. [320] 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 preceeding 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. [19, 58, 297, 320]. 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. 41.12.

41.7.4 *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 trade-off 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 trade-offs were studied in Refs. [319, 320] with both pedagogical and physically-motivated examples.

41.7.5 *Aleatoric and epistemic uncertainty*

In the machine learning and risk assessment literature, uncertainty is often characterized in terms of *aleatoric* and *epistemic* uncertainty [321–324]. 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. [323], states that "roughly speaking, aleatoric (aka 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 (aka 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. [325]) 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[6]. 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

[6] 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.

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, natural language, *etc.*). 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. [324] 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 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).

41.7.6 *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 $\mathcal{L}_{MSE} = (y - f_\phi(x))^2$ 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. 41.3.1.2). With this view in mind, 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

$$p(y|x,\mathcal{D}) = \int d\phi p(\phi|\mathcal{D}) p(y|x,\phi). \tag{41.76}$$

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. [326] provides a different perspective on BMA analyzed in what are referred to as the $\mathcal{M}$-open and $\mathcal{M}$-closed settings [326]. 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 [326]. Ref. [327] 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 [328], which uses dropout ensembling, and deep ensembles [329], 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 [330].

41.7.7 *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 41.3.5) and deep generative models (see Sec. 41.5.4). 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. 41.5.1) are an example, but the form of the models is limited to Gaussian posteriors. In Sec 41.3.5 we discussed approaches to model $p(y|x)$ using conditional density estimation [51,56,57]. 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 $\hat{f}_{MSE}(x) \approx p(y=1|x)$ (see Eq. 41.10). Incorporating the dependence on the nuisance parameter, then connects to the likelihood-ratio trick (see Eq. 41.12), approaches to simulation-based inference that involve learning the likelihood-ratio, and the parameterized approaches described in Sec. 41.7.3.

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. 41.7.5 and Sec. 41.7.1, 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. 41.7.1. 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. [58, 320]). Unfortunately, calibration in the context of density estimation is more challenging. This connects to topics and challenges in anomaly detection (see Sec. 41.3.4).

41.8 Infrastructure for deployment in experiments

The software and computing needs of training a machine learning model are different than those encountered when it deployed for use. In machine learning jargon, the two stages are often referred to as *training* and *inference*, where inference might refer to making a prediction for y given x in a classification or regression problem. 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` provides an open source format for many types of models and is is widely supported and can be found in many frameworks, tools, and hardware. This is important when integrating a trained model into the large 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 [81, 108, 331–340]. For applications where latency is a key concern (eg triggering at collider experiments), various accelerators have been investigated [167, 171, 341–355]. In addition, some solutions for deployment of ML models involve deployment in the cloud [356, 357].

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

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

[10] 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.

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

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

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

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

[15] 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 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018," volume 80 of *Proceedings of Machine Learning Research*, 540–548, PMLR (2018), URL http://proceedings.mlr.press/v80/belkin18a.html.

[16] 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.

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

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

[19] J. Brehmer *et al.*, Phys. Rev. **D98**, 5, 052004 (2018), [arXiv:1805.00020].

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

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

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

[23] 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.

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

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

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

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

[28] https://en.wikipedia.org/wiki/Cluster_analysis.

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

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

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

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

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

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

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

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

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

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

[39] M. Farina, Y. Nakai and D. Shih, Physical Review D **101**, 7 (2020), ISSN 2470-0029, URL http://dx.doi.org/10.1103/PhysRevD.101.075021.

[40] T. Heimel *et al.*, SciPost Physics **6**, 3 (2019), ISSN 2542-4653, URL http://dx.doi.org/10.21468/SciPostPhys.6.3.030.

[41] 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).

[42] 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.

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

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

[45] 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.

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

[47] E. M. Metodiev, B. Nachman and J. Thaler, Journal of High Energy Physics **2017**, 10 (2017), ISSN 1029-8479, URL http://dx.doi.org/10.1007/JHEP10(2017)174.

[48] 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.

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

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

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

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

[53] 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.

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

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

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

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

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

[59] C. Modi, F. Lanusse and U. Seljak, "Flowpm: Distributed tensorflow implementation of the fastpm cosmological n-body solver," (2020), [arXiv:2010.11847].

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

[61] U. Seljak *et al.*, JCAP **12**, 009 (2017), [arXiv:1706.06645].

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

[63] K. Datta, D. Kar and D. Roy (2018), [arXiv:1806.00433].

[64] M. Bellagente *et al.* (2019), [arXiv:1912.00477].

[65] N. D. Gagunashvili (2010), [arXiv:1004.2006].

[66] A. Glazov (2017), [arXiv:1712.01814].

[67] M. Bellagente *et al.* (2020), [arXiv:2006.06685].

[68] M. Vandegar *et al.*, in A. Banerjee and K. Fukumizu, editors, "Proceedings of The 24th International Conference on Artificial Intelligence and Statistics," volume 130 of *Proceedings of Machine Learning Research*, 2107–2115, PMLR (2021), [arXiv:2011.05836], URL https://proceedings.mlr.press/v130/vandegar21a.html.

[69] P. Baroň (2021), [arXiv:2104.03036].

[70] A. Andreassen *et al.* (2021), [arXiv:2105.04448].

[71] P. Komiske, W. P. McCormack and B. Nachman (2021), [arXiv:2105.09923].

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

[73] M. Arratia *et al.* (2021), [arXiv:2109.13243].

[74] T. A. Le, A. G. Baydin and F. Wood, in "Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017," volume 54 of *Proceedings of Machine Learning Research*, 1338–1348, PMLR, Fort Lauderdale, FL, USA (2017), [arXiv:1610.09900].

[75] A. G. Baydin *et al.* (2018), [arXiv:1807.07706].

[76] A. G. Baydin *et al.*, International Conference for High Performance Computing, Networking, Storage and Analysis, SC arXiv:1907.03382 (2019), ISSN 21674337, [arXiv:1907.03382].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[137] J. García-Méndez *et al.* (2021), [arXiv:2107.13654].

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

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

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

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

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

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

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

[145] J. Shlomi *et al.* (2020), [arXiv:2008.02831].

[146] M. J. Dolan and A. Ore (2020), [arXiv:2012.00964].

[147] M. J. Fenton *et al.* (2020), [arXiv:2010.09206].

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

[149] V. Mikuni and F. Canelli (2021), [arXiv:2102.05073].

[150] A. Shmakov *et al.* (2021), [arXiv:2106.03898].

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

[152] Technical Report ATL-PHYS-PUB-2020-014, CERN, Geneva (2020), URL https://cds.cern.ch/record/2718948.

[153] G. Louppe *et al.*, Journal of High Energy Physics **2019**, 1, 57 (2019), ISSN 10298479, [arXiv:1702.00748].

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

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

[156] X. Ju *et al.*, 33rd Annual Conference on Neural Information Processing Systems (2020), [arXiv:2003.11603].

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

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

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

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

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

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

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

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

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

[166] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[183] 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.

[184] B. E. Boser, I. M. Guyon and V. N. Vapnik, in "Proceedings of the fifth annual workshop on Computational learning theory," 144–152 (1992).

[185] C. Cortes and V. Vapnik, Machine learning **20**, 3, 273 (1995).

[186] H. Drucker *et al.*, Advances in neural information processing systems **9**, 155 (1997).

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

[188] C. K. Williams, Advances in neural information processing systems 295–301 (1997).

[189] J. Lee *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=B1EA-M-0Z.

[190] 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.

[191] T. Hofmann, B. Schölkopf and A. J. Smola, The Annals of Statistics 1171–1220 (2008).

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

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

[194] S. Ambikasaran *et al.*, IEEE transactions on pattern analysis and machine intelligence **38**, 2, 252 (2015).

[195] J. R. Gardner *et al.*, arXiv preprint arXiv:1809.11165 (2018).

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

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

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

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

[200] "Xgboost," https://xgboost.readthedocs.io/.

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

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

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

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

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

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

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

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

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

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

[211] 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.

[212] 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.

[213] 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.

[214] M. Lin, Q. Chen and S. Yan, arXiv preprint arXiv:1312.4400 (2013).

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

[216] K. He *et al.*, in "2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)," 770–778 (2016).

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

[218] A. Bietti, L. Venturi and J. Bruna, arXiv preprint arXiv:2106.07148 (2021).

[219] M. M. Bronstein *et al.*, arXiv preprint arXiv:2104.13478 (2021).

[220] K. Simonyan and A. Zisserman, CoRR **abs/1409.1556** (2015).

[221] 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.

[222] M. collaboration, Journal of Instrumentation **12**, 03, P03011 (2017), URL https://doi.org/10.1088/1748-0221/12/03/p03011.

[223] 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.

[224] G. Huang *et al.*, in "2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)," 2261–2269 (2017).

[225] D. Rumelhart, G. Hinton and R. Williams .

[226] Y. Bengio, P. Simard and P. Frasconi, Neural Networks, IEEE Transactions on **5**, 2, 157 (1994).

[227] 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.

[228] S. Hochreiter and J. Schmidhuber, Neural Computation **9**, 8, 1735 (1997).

[229] R. Socher *et al.*, in "ICML," (2011).

[230] R. Socher *et al.*, in "Proceedings of the 2011 conference on empirical methods in natural language processing," 151–161 (2011).

[231] X. Chen *et al.*, in "Proceedings of the 2015 conference on empirical methods in natural language processing," 793–798 (2015).

[232] C. Olah and S. Carter, Distill (2016), URL http://distill.pub/2016/augmented-rnns.

[233] 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.

[234] A. Vaswani *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/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[235] R. Bommasani *et al.*, arXiv preprint arXiv:2108.07258 (2021).

[236] P. Battaglia *et al.*, arXiv (2018), URL https://arxiv.org/pdf/1806.01261.pdf.

[237] 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).

[238] C. R. Qi *et al.*, in "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)," (2017).

[239] C. Qi *et al.*, in "NIPS," (2017).

[240] 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.

[241] 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.

[242] Y. Wang *et al.*, ACM Transactions on Graphics **38** (2018).

[243] 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.

[244] 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.

[245] N. Choma *et al.* (IceCube) (2018), [arXiv:1809.06166].

[246] 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.

[247] 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.

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

[249] 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.

[250] 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.

[251] 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.

[252] 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.

[253] L. Dinh, D. Krueger and Y. Bengio, 3rd International Conference on Learning Representations, ICLR 2015 - Workshop Track Proceedings (2015), [arXiv:1410.8516].

[254] 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.

[255] 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).

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

[257] 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.

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

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

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

[261] 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).

[262] 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.

[263] 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.

[264] 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).

[265] 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.

[266] 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.

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

[268] 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.

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

[270] C. Durkan *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," 7509–7520 (2019).

[271] 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.

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

[273] 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.

[274] A. Hyvärinen and P. Dayan, Journal of Machine Learning Research **6**, 4 (2005).

[275] Y. Song and S. Ermon, in Wallach *et al.* [358], 11895–11907, URL https://proceedings.neurips.cc/paper/2019/hash/3001ef257407d5a371a96dcd947c7d93-Abstract.html.

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

[277] 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.

[278] D. P. Kingma and J. Ba, arXiv e-prints arXiv:1412.6980 (2014), [arXiv:1412.6980].

[279] 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.

[280] Y. Yao, L. Rosasco and A. Caponnetto, Constructive Approximation **26**, 2, 289 (2007).

[281] L. Prechelt, in "Neural Networks: Tricks of the trade," 55–69, Springer (1998).

[282] A. Krizhevsky, I. Sutskever and G. Hinton, Neural Information Processing Systems **25** (2012).

[283] 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.

[284] S. Ioffe and C. Szegedy, CoRR **abs/1502.03167** (2015), [arXiv:1502.03167], URL http://arxiv.org/abs/1502.03167.

[285] J. L. Ba, J. R. Kiros and G. E. Hinton, "Layer normalization," (2016), [arXiv:1607.06450].

[286] D. Ulyanov, A. Vedaldi and V. Lempitsky, "Instance normalization: The missing ingredient for fast stylization," (2017), [arXiv:1607.08022].

[287] Y. Wu and K. He, "Group normalization," (2018), [arXiv:1803.08494].

[288] 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.

[289] O. Russakovsky *et al.*, International Journal of Computer Vision (IJCV) **115**, 3, 211 (2015).

[290] M. Cordts *et al.*, in "Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)," (2016).

[291] L. Yi *et al.*, SIGGRAPH Asia (2016).

[292] A. Radford *et al.* (2018), URL https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.

[293] L. Fei-Fei, R. Fergus and P. Perona, IEEE Transactions on Pattern Analysis and Machine Intelligence **28**, 4, 594 (2006).

[294] H. Larochelle, D. Erhan and Y. Bengio, in "Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2," AAAI'08, 646–651, AAAI Press (2008), ISBN 9781577353683.

[295] M. Palatucci *et al.*, in Y. Bengio *et al.*, editors, "Advances in Neural Information Processing Systems," volume 22, Curran Associates, Inc. (2009), URL https://proceedings.neurips.cc/paper/2009/file/1543843a4723ed2ab08e18053ae6dc5b-Paper.pdf.

[296] R. Socher *et al.*, in "Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1," NIPS'13, 935–943, Curran Associates Inc., Red Hook, NY, USA (2013).

[297] T. Dorigo and P. De Castro Manzano (2020), [arXiv:2007.09121].

[298] R. Barate *et al.* (ALEPH), Phys. Lett. B **412**, 173 (1997).

[299] 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.

[300] H. Edwards and A. Storkey, arXiv preprint arXiv:1511.05897 (2015).

[301] Y. Ganin and V. Lempitsky, in "International conference on machine learning," 1180–1189, PMLR (2015).

[302] H. Ajakan *et al.*, ArXiv e-prints (2014), [arXiv:1412.4446].

[303] G. Kasieczka and D. Shih, Phys. Rev. Lett. **125**, 12, 122001 (2020), [arXiv:2001.05310].

[304] C. Shimmin *et al.*, Phys. Rev. **D96**, 7, 074034 (2017), [arXiv:1703.03507].

[305] J. Stevens and M. Williams, JINST **8**, P12013 (2013), [arXiv:1305.7248].

[306] J. Dolen *et al.*, JHEP **05**, 156 (2016), [arXiv:1603.00027].

[307] I. Moult, B. Nachman and D. Neill, JHEP **05**, 002 (2018), [arXiv:1710.06859].

[308] L. Bradshaw *et al.* (2019), [arXiv:1908.08959].

[309] ATL-PHYS-PUB-2018-014 (2018), URL http://cds.cern.ch/record/2630973.

[310] L.-G. Xia, Nucl. Instrum. Meth. **A930**, 15 (2019), [arXiv:1810.08387].

[311] C. Englert *et al.*, Eur. Phys. J. **C79**, 1, 4 (2019), [arXiv:1807.08763].

[312] S. Wunsch *et al.* (2019), [arXiv:1907.11674].

[313] A. Rogozhnikov *et al.*, JINST **10**, 03, T03002 (2015), [arXiv:1410.4140].

[314] C. Collaboration, Machine Learning: Science and Technology (2020).

[315] J. M. Clavijo, P. Glaysher and J. M. Katzy (2020), [arXiv:2005.00568].

[316] G. Kasieczka *et al.* (2020), [arXiv:2007.14400].

[317] O. Kitouni *et al.* (2020), [arXiv:2010.09745].

[318] A. Ghosh and B. Nachman (2021), [arXiv:2109.08159].

[319] P. Baldi *et al.*, Eur. Phys. J. **C76**, 5, 235 (2016), [arXiv:1601.07913].

[320] A. Ghosh, B. Nachman and D. Whiteson (2021), [arXiv:2105.08742].

[321] 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.

[322] 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.

[323] E. Hüllermeier and W. Waegeman, CoRR **abs/1910.09457** (2019), URL http://arxiv.org/abs/1910.09457.

[324] 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.

[325] 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.

[326] Y. Yao *et al.*, Bayesian Analysis **13**, 3 (2018), ISSN 1936-0975, URL http://dx.doi.org/10.1214/17-BA1091.

[327] J. Snoek *et al.*, in Wallach *et al.* [358], 13969–13980, URL https://proceedings.neurips.cc/paper/2019/hash/8558cb408c1d76621371888657d2eb1d-Abstract.html.

[328] Y. Gal and Z. Ghahramani, in M. Balcan and K. Q. Weinberger, editors, "Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016," volume 48 of *JMLR Workshop and Conference Proceedings*, 1050–1059, JMLR.org (2016), URL http://proceedings.mlr.press/v48/gal16.html.

[329] 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.

[330] D. P. Kingma, T. Salimans and M. Welling, CoRR **abs/1506.02557** (2015), URL http://arxiv.org/abs/1506.02557.

[331] G. C. Strong (2020), [arXiv:2002.01427].

[332] V. V. Gligorov and M. Williams, JINST **8**, P02013 (2013), [arXiv:1210.6861].

[333] D. W. III *et al.* (2017), URL https://dl4physicalsciences.github.io/files/nips_dlps_2017_3.pdf.

[334] D. Bourgeois, C. Fitzpatrick and S. Stahl (2018), [arXiv:1808.00711].

[335] J. Alimena, Y. Iiyama and J. Kieseler (2020), [arXiv:2004.10744].

[336] C. Balázs *et al.* (DarkMachines High Dimensional Sampling Group) (2021), [arXiv:2101.04525].

[337] F. Rehm *et al.* (2021), [arXiv:2103.10142].

[338] C. Mahesh *et al.*, in "34th Conference on Neural Information Processing Systems," (2021), [arXiv:2104.06622].

[339] S. Amrouche *et al.* (2021), [arXiv:2105.01160].

[340] P. Goncharov *et al.*, in "24th International Scientific Conference of Young Scientists and Specialists," (2021), [arXiv:2109.08982].

[341] J. Duarte *et al.*, JINST **13**, 07, P07027 (2018), [arXiv:1804.06913].

[342] J. Ngadiuba *et al.*, Mach. Learn.: Sci. Tech. **2**, 1, 015001 (2020), [arXiv:2003.06308].

[343] S. Summers *et al.*, JINST **15**, 05, P05026 (2020), [arXiv:2002.02534].

[344] J. Krupa *et al.* (2020), [arXiv:2007.10359].

[345] L. R. M. Mohan *et al.* (2020), [arXiv:2008.09210].

[346] S. Carrazza, J. M. Cruz-Martinez and M. Rossi (2020), [arXiv:2009.06635].

[347] D. S. Rankin *et al.*, 2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC) 38 (2020), [arXiv:2010.08556].

[348] M. Rossi, S. Carrazza and J. M. Cruz-Martinez (2020), [arXiv:2012.08221].

[349] T. Aarrestad *et al.* (2021), [arXiv:2101.05108].

[350] B. Hawks *et al.* (2021), [arXiv:2102.11289].

[351] T. Teixeira, L. Andrade and J. M. de Seixas (2021), [arXiv:2103.12467].

[352] T. M. Hong *et al.* (2021), [arXiv:2104.03408].

[353] G. Di Guglielmo *et al.* (2021), [arXiv:2105.01683].

[354] M. Migliorini *et al.* (2021), [arXiv:2105.04428].

[355] E. Govorkova *et al.* (2021), [arXiv:2108.03986].

[356] V. Kuznetsov, L. Giommi and D. Bonacorsi (2020), [arXiv:2007.14781].

[357] O. Sunneborn Gudnadottir *et al.*, EPJ Web Conf. **251**, 02054 (2021), [arXiv:2109.00264].

[358] 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* (2019), URL `https://proceedings.neurips.cc/paper/2019`.