Machine learning and its interpretability

Luisa Lucie-Smith[1, *]

[1]*Max-Planck-Institut für Astrophysik, Karl-Schwarzschild-Str. 1, 85748 Garching, Germany*

(Dated: November 1, 2023)

Lecture notes for the *Lecture Series on Cosmology* held on November 2nd 2023 at MPA in Garching.

CONTENTS

* luisals@mpa-garching.mpg.de

I. INTRODUCTION

The basic idea behind machine learning algorithms is to automatically build a model that can describe the data given a set of examples, known as the *training set* [1]. Machine learning is usually employed for problems that are so highly non-linear that they can not be solved using traditional (analytic or numerical) statistical techniques. The advantage of machine learning is that it can automatically build a model up to an arbitrary level of complexity. Provided the training set is a representative sub-sample of the data, the trained model can then be used to make predictions on new unseen data.

The three main categories of machine learning techniques are *supervised learning*, where the training set is given by a set of input features and their corresponding label, *unsupervised learning* where the training set consists of features with no corresponding label, and *reinforcement learning* which also does not need labels for each data instance. Typically supervised learning is used for classification problems, where the training samples belong to two or more classes, or regression problems, where the output consists of continuous variables. Unsupervised learning is usually used in clustering problems, where the aim is to group similar samples in the data, or density estimation. In addition, *semi-supervised* learning algorithms, which involve a mixture of labelled and unlabelled training samples, can also be used in problems such as image recognition. Reinforcement learning instead involves a system that is rewarded or penalised at specific points during learning, based on how it performs. It must then learn by itself what is the best strategy to optimise a global task (e.g. earn the most reward over time). This approach has been taken to teach robots how to walk, or to train machines how to play games (e.g. AlphaGo [6]).

Machine learning is utilized for a variety of different tasks. Some of the most common tasks in Astrophysics include:

- Classification: The training samples belong to two or more discrete classes, and the learning algorithm is asked to produce $y = f(\mathbf{x})$ where y denotes the numeric code of a class or the probability distribution over classes. An example of a classification task in astrophysics is transient classification, where the input is the observed spectrum and the output is a numeric code identifying the supernova type (Ia, II,..).

- Regression: This case is similar to the classification task, except the output of the learning algorithm consists of a continuous variable instead of a class. An example of this is predicting the redshift of a galaxy from spectroscopic or photometric data.

- Denoising: In this type of task, the machine learning algorithm is given as input a noisy image $\tilde{\mathbf{x}}$ obtained by an unknown source of noisy, and asked to predict the 'denoised' or clean example $\mathbf{x}$ from its corrupted version $\tilde{\mathbf{x}}$, or more generally predict the conditional probability distribution $p(\mathbf{x}|\tilde{\mathbf{x}})$.

- Density estimation: In the density estimation problem, the machine learning algorithm is asked to learn a function p_{model}, where $p_{\text{model}}(x)$ can be interpreted as a probability density function (if x is continuous) or a probability mass function (if x is discrete) on the space that the examples were drawn from. To do such a task well, the algorithm needs to learn the structure of the data it has seen. Note that most other tasks also require the learning algorithm to implicitly capture the structure of the probability distribution; here, we ask the learning model to return that probability distribution explicitly.

A. Interpretability

As physicists, our goal is to build mathematical models that can be solved analytically or numerically to provide useful predictions (deterministically or probabilistically) about phenomena of interest. These models typically come from an explicit set of assumptions and physical principles, and depend on ingredients that are grounded in domain knowledge. For example, cosmologists can model the linear matter power spectrum in terms of cosmological parameters given knowledge of linear

[1] For good introductions to the field of machine learning see Abu-Mostafa *et al.* [1], Hastie *et al.* [2], MacKay [3], Bishop [4], Murphy [5].

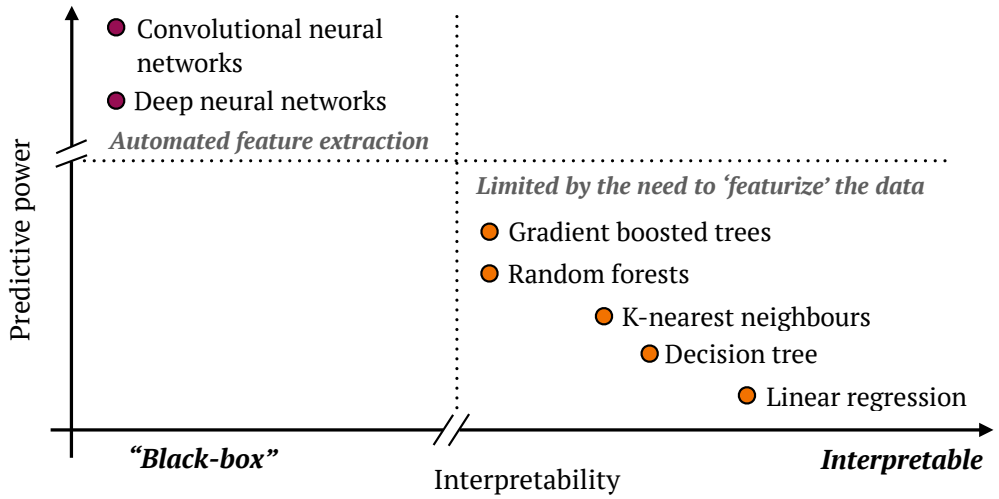


FIG. 1. The interpretability vs predictive power trade-off. The best performing models are so complex that they lack transparency and are hence considered to be "black-boxes"; simpler algorithms such as linear regression models are straightforward to interpret as they are based on simple and smooth relationships between the inputs and outputs, but are limited in complexity.

theory. However, when building machine learning models, this is often not the case. Understanding the inner workings of machine learning models, especially in the context of deep learning, remains a challenge. This means that one does not know how and why certain predictions are made by the machine learning algorithm, and what underlying ingredients does that model depend on.

Often, the best performing machine learning models are so complex that they lack of transparency and are hence considered to be "black-boxes". On the other hand, simpler algorithms such as linear regression models are straightforward to interpret as they are based on simple and smooth relationships between the inputs and outputs, but are limited in complexity. This leads to an accuracy vs. interpretability trade-off in machine learning (Fig. 1); black-box models, such as convolutional neural networks, have the power to extract highly-complex, non linear pattern in the data, thus yielding great accuracy. This complexity is made possible thanks to the millions of parameters that make up these models, thus making it very difficult to understand what information the algorithm detects, and how that information translates into the models' final predictions. This is why they are known as "black-box algorithms". In science, we require the ability to explain how and why certain predictions are made by a model, thus making the field of interpretability of machine learning algorithms of primary importance. Developing techniques to turn "black-box" algorithms into interpretable ones is essential for machine learning applications to cosmology; ultimately, it will allow us to interpret machine learning results in terms of the underlying physics. Interpretability serves the dual purpose of understanding and trust: we learn about the underlying factors required to model the output of interest, while ensuring that the model's learning is trustworthy and robust.

Interpretability is more formally defined as a way to account for why a certain machine learning model reaches a particular decision or prediction. However, this is not quite enough for machine learning models to acquire the characteristics of (good) physical models. One must also be also to map this account onto existing knowledge in the relevant science domain; this is formally known as *explainability*. For example, one may develop an interpretability tool that allows you to visualize which patterns are extracted in the layers of a neural network, but these may be hard to translate into meaningful *physical* features within the physics domain. The goal is therefore to be able to interpret the results of the machine learning models *and* explain them in terms of the physics they represent.

II. A RECIPE FOR TRAINING MACHINE LEARNING MODELS

Nearly all machine learning algorithms are built following a very similar recipe: introduce a model (for example a neural network), obtain a representative dataset of examples, write down the appropriate likelihood (or loss) function, and optimize the parameters of the model to minimize the latter. Recognizing that most machine learning algorithms can be described using this recipe helps to see the different algorithms as part of a taxonomy of methods for doing related tasks that work for similar reasons, rather than as a long list of algorithms that each have separate justifications. We will begin with describing each of these ingredients, using a linear regression algorithm as a simple illustrative example. We will then demonstrate how these principles can be applied to more complicated learning algorithms, such as decision trees and convolutional neural networks.

A. The data

The most important ingredient for machine learning to work is to have enough data samples. As a general rule of thumb, if one has less than a few hundred examples, it is very likely more data will be needed. The data should be split into three subsets of examples: the training set, the validation set and the test set. The training set is used to train the machine learning model; the validation set to track the performance of the algorithm on unseen data; the test set is set aside and left completely blind of entire training pipeline and used to test the final model. The most important role of the validation set is to tune the *hyperparameters* of the model: these are parameters of the model that are not tuned during the algorithm's learning process but must be manually set a priori. Generally, these are parameters that describe high-level characteristics of the model and need to be optimized for any given training set. Examples of hyperparameters in decision trees are the maximum depth of the tree, or for neural networks the number of hidden layers. To optimize the hyperparameters, one possible strategy is to construct grids of values for each hyperparameter, consider all possible combinations of parameters and choose the setting which performs best on the validation set according to some evaluation metric. The fact that the choice of hyperparameters is tested on an independent validation, rather than on the training set, avoids overfitting to the latter. The test set is then used to test how well the model generalizes, once both the optimal hyperparameters and learning parameters have been fixed.

For most 'traditional' machine learning algorithms, it can be difficult to learn patterns in the training set if the input data is made of noisy and high-dimensional data. Particularly now in the era of "Big Data", we are faced with large amounts of high-dimensional data. Feature extraction and feature selection are two powerful tools which can address these issues by projecting the original high-dimensional space into a new low-dimensional feature space (feature extraction) and by selecting only a subset of informative features to construct the algorithm (feature selection) (Li *et al.* [7]). Both tools can provide significant improvements in the algorithm's performance[2], as well as requiring lower computational and memory costs. As we will see later on, deep learning models such as convolutional neural networks have the advantage of avoiding the featurization step altogether, since they automatically perform feature extraction during training.

B. The likelihood

A machine learning model (e.g. a linear regression model or a neural network) can be viewed as a probabilistic model $p_\theta(\mathbf{y}|\mathbf{x}, \theta)$, where given an input $\mathbf{x}$, a neural network assigns a probability to each possible output $\mathbf{y}$, using the set of parameters θ. The training set encodes the prior of the problem, such that $\mathcal{D} = \{x_i, y_i\} \sim p(x, y) = p(y|x)p(x)$. The parameters θ are typically learnt via maximum likelihood estimation (MLE): given a set of training examples $\mathcal{D} = \{x_i, y_i\}_{i=1}^{d}$, the optimal weights are those that maximize the likelihood $p\left(\mathbf{y} \mid \mathbf{x}, \theta\right)$. In other words, the maximum likelihood estimator for θ is defined

[2] Note that whilst feature extraction is indispensable for all machine learning algorithms (with the exception of convolutional neural networks), not all require feature selection; some are robust to a large number of uninformative features.

as

$$\hat{\theta} = \arg\max_{\theta} p\left(\mathbf{y} \mid \mathbf{x}, \theta\right) = \arg\max_{\theta} \prod_{i=1}^{d} p\left(y_i \mid x_i, \theta\right) = \arg\max_{\theta} \sum_{i=1}^{d} \ln p\left(y_i \mid x_i, \theta\right). \tag{1}$$

This maximization is typically turned into an (equivalent) minimization problem, such that

$$\hat{\theta} = \arg\min_{\theta} \left[-\sum_{i=1}^{d} \ln p\left(y_i \mid x_i, \theta\right) \right], \tag{2}$$

where the negative log-likelihood is more generally called the loss function $\mathcal{L}$ in the machine learning community. The likelihood must be chosen so that it best represents the data. A typical choice is to assume that the training data are Gaussian distributed, $y_i = \mathcal{N}(\hat{y}_i, \sigma^2)$, and the likelihood is given by

$$p\left(y_i \mid x_i, \theta\right) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2} \left(\frac{y_i - \hat{y}_i(x_i, \theta)}{\sigma} \right)^2 \right] \tag{3}$$

which in turn gives

$$\mathcal{L} = -\sum_{i=1}^{d} \ln p\left(y_i \mid x_i, \theta\right) = \frac{d}{2} \ln(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^{d} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{d} (y_i - \hat{y}_i)^2, \tag{4}$$

where in the last line we have ignored all terms that do not depend on θ. If we rescale the loss by $1/d$ so that it is insensitive to the training set size, the loss becomes

$$\mathcal{L} = \frac{1}{d} \sum_{i=1}^{d} ||y_i - \hat{y}_i||^2 \tag{5}$$

which is equivalent to the well-known mean squared error loss that is often adopted (without justification) in machine learning. Therefore, we immediately see that maximizing the log-likelihood with respect to θ yields the same estimate of the parameters θ as does minimizing the mean squared error. The two criteria have different values but the same location of the optimum. This justifies the use of the MSE as a maximum likelihood estimation procedure.

C. Optimization

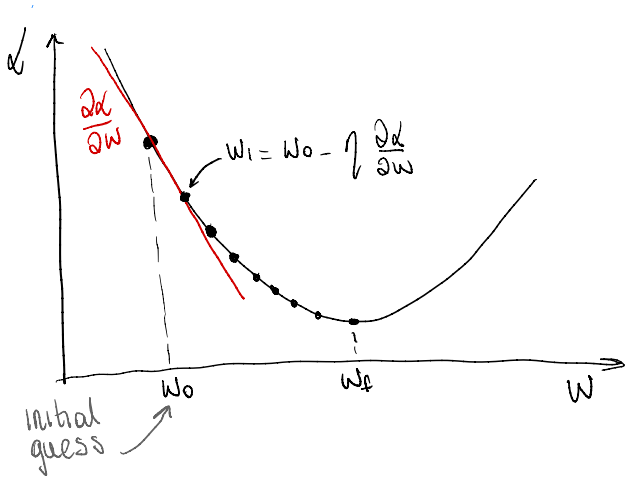
The second fundamental ingredient to obtain a working model is the optimization strategy. The loss function measures how good the estimated output $\hat{y}$ is, and tells the optimization method how to make it better. The optimization procedure is what changes the model in order to improve the loss function (i.e. to improve the estimation).

Training a neural network involves finding the set of parameters θ which minimize the loss function, as we wrote in Eq. (2). The best estimate for θ can be found by solving $\partial\,\mathcal{L}(\theta)/\partial\theta = 0$. The way this is most often done is through an iterative process: we start with a random set of weight values which get iteratively updated until one converges to the final set of weights that minimize the loss function. The method used to update the weights at each iteration is *gradient descent*. A gradient descent algorithm starts with a random point on a function and travels down in the direction of steepest descent in steps until it reaches the lowest point of the function.

The direction of steepest descent of the loss function at point w_0 is given by

$$w_1 = w_0 - \eta \left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w_0}, \tag{6}$$

where η is the learning rate, which measures how much to change the weight value with respect to the gradient.



Most optimization algorithms utilized in machine learning are based on variants of vanilla gradient descent. For example, mini-batch gradient descent is a popular algorithm used in neural networks where the model parameters (or weights) are updated based on small subsets (or batches) of the entire training data. In other words, the dataset is divided into various batches and after every batch, the parameters are updated. The frequent updates of model parameters implies convergence of the model in less time and requires less amount of memory.

Most gradient descent algorithms require choosing an optimum value of the learning rate. If the learning rate is too small, gradient descent will take forever to converge because the weights are updated in tiny steps at a time. If the learning rate is too high, the weights get updated in too large steps that overshoots the minimum and convergence is never reached. Another issue can be related to having a constant learning rate for all the parameters, since there may be some parameters which we may not want to change at the same rate. These issues have motivated the community to develop different optimizers to overcome these issues, include optimizers that have the learning rate for each parameter and at every time step (e.g. Adagrad [8] or Adam [9]).

D. The model

Finally, the last (or first) ingredient required to build a machine learning model is to choose which model we would like to train. This can be done by making use of inductive biases. Every machine learning model requires some type of architecture design and possibly some initial assumptions about the data we want to analyze. In Bayesian terms, inductive bias shows itself in the form of the prior distributions. For example, the k-Nearest Neighbors (kNN) algorithm assumes that entities belonging to a particular category should appear near each other, and those that are part of different groups should be distant. In other words, we assume that similar data points are clustered near each other away from the dissimilar ones. For convolutional neural networks, the inductive bias is on locality i.e features of the data can be constructed using local pixels and then combined hierarchically. In linear regression, we assume that the output is linearly dependent on the input variables. Therefore, the resulting model linearly fits the training data.

We will now take a linear regression model to provide an illustration of the recipe for training models that we just described.

1. An example: linear regression model

Let's take a linear regression model as a simple example to show how to apply the recipe learnt so far in practice; remember that the basic principles are still valid when training complex deep learning models. The goal of a linear regression model is to find a linear model that explains the target $\mathbf{y}$ given a set of inputs $\mathbf{X}$. Given a training set $\{\mathbf{x}_{1:n}, \mathbf{y}_{1:n}\}$, the goal is to find the

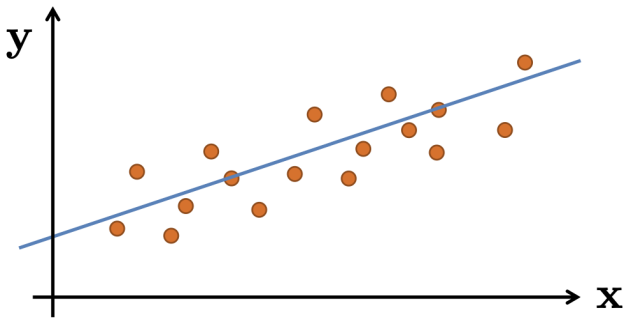


FIG. 2. A linear regression model fitted to a set of $\{\mathbf{x}_{1:n}, \mathbf{y}_{1:n}\}$ points shown as orange scatter points.

optimal parameters $\hat{w}$, also known as weights, of a model f_w. The trained model is then used as a predictor for unseen data $\hat{\mathbf{y}}_{n+1} = f_{\hat{\theta}}(\mathbf{x}_{n+1})$. A linear model is expressed in the form

$$\hat{y}_i = f_\theta(x_i) = \sum_{j=1}^{d} x_{ij} w_j = \mathbf{x}_i \mathbf{w} \tag{7}$$

where d is the input dimensionality, x_{ij} are the inputs, or "features", of the i-th sample, and $\hat{y}_i$ is the predicted output. For example, you may want to write the temperature of the building (y) as a weighted function of the outside temperature (x_1), the size of the building (x_2) and the sun exposure (x_3). The model parameters are the weights assigned to each independent variable (w_1, w_2, w_3). If we assume a Gaussian likelihood, the loss function we wish to minimize becomes the mean squared error,

$$\mathcal{L}(\mathbf{w}) = \frac{1}{d} \sum_{i=1}^{d} (\hat{y}_i - y_i)^2 = \frac{1}{d} \sum_{i=1}^{d} (\mathbf{x}_i \mathbf{w} - y_i)^2 . \tag{8}$$

In matrix notation, this is

$$\mathcal{L}(\mathbf{w}) = (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y}). \tag{9}$$

In linear regresson, we can solve this equation explicitly. However, note that this is only possible for a handful of models; by contrast, we can apply gradient descent to any model for which we can compute the gradient. To minimize the loss, we can then simply solve for where its gradient is 0:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y}) = 2\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\mathbf{X}^T\mathbf{y} = 0. \tag{10}$$

so that

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{y}. \tag{11}$$

E. Generalization

The above recipe tackled the issue of finding the optimal parameters of the model to describe the set of examples that make up the training set. The central challenge in machine learning is that our algorithm must perform well on new, previously unseen inputs and not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization. We can control whether a model is more likely to overfit or underfit by altering its capacity (or complexity). Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may

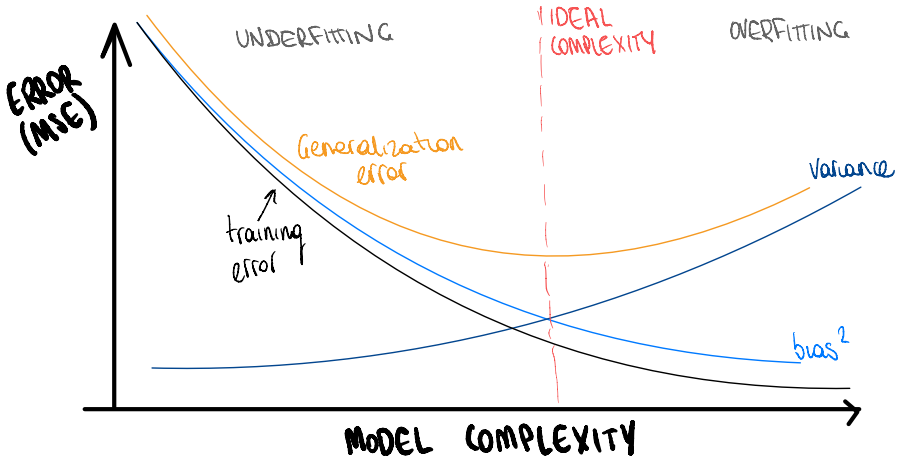


FIG. 3. The error as a function of model complexity. A model that under-fits the data generally has low complexity and its error budget involves a high bias and a low variance. A model that over-fits the data is too flexbile (or has high capacity) and its error budget involves a high variance and a low bias. Finding the right balance of bias and variance is key to creating an effective and accurate model.

struggle to fit the training set because they do not have enough flexibility, whereas models with high capacity can overfit by memorizing properties of the training set that do not generalize or apply to the test set. An illustration of the behaviour of the error as a function of model capacity is shown in Fig. 3.

Specifically when training using mean squared error, generalization is tightly related to the machine learning's trade-off in their ability to minimize bias and variance in the predictions [2, 10]. The bias is the difference between the average prediction of the model and the correct value we are trying to predict, whereas the variance is the variability of the model's prediction for a given correct value (or sample). A model that under-fits the data generally has a high bias and a low variance, whereas one that over-fits has a low bias and a high variance. The key is to find the right balance in the complexity of the model which neither over-fits nor under-fits the data; this trade-off in complexity gives rise to the trade-off between bias and variance. An understanding of these errors can help with choosing the appropriate machine learning algorithm for a given problem and to build accurate models.

III. DECISION TREES

One special family of machine learning algorithms that provide excellent accuracy with very high interpretability are ensemble methods, such as random forests and gradient boosted trees. In this section, I will first review decision trees, as they form the building block of these ensemble models, and then move on to describing random forests and gradient boosted trees.

As previously mentioned, nearly all deep learning algorithms can be described as particular implementations of a simple recipe: obtain a representative dataset, a loss function, an optimization procedure and a model. For example, the linear regression algorithm combines a dataset consisting of $\mathbf{X}$ and $\mathbf{y}$, the model $p(y|x) = \mathcal{N}(y; xw + b, 1)$, the loss function $-\log p(y|x)$ and in most cases, the optimization algorithm defined by solving for where the gradient of the cost is minimized (e.g. gradient descent). Most machine learning algorithms make use of this recipe, though it may not be immediately obvious. If a machine learning algorithm seems especially unique, it can usually be understood as using a special-case optimizer. For example, decision trees require special-case optimizers because their loss functions are not differentiable throughout making them inappropriate for minimization by gradient-based optimizers.

A decision tree is a supervised learning method which predicts the output of a sample by following a set of simple decision rules inferred from the features of the data, $\mathbf{X}$ [2, 11–13]. A tree is formed by a set of nodes, each with its own decision rule of the form $X_i \leq n$, where X_i is the feature which makes the split and n is some value of that feature that determines the split

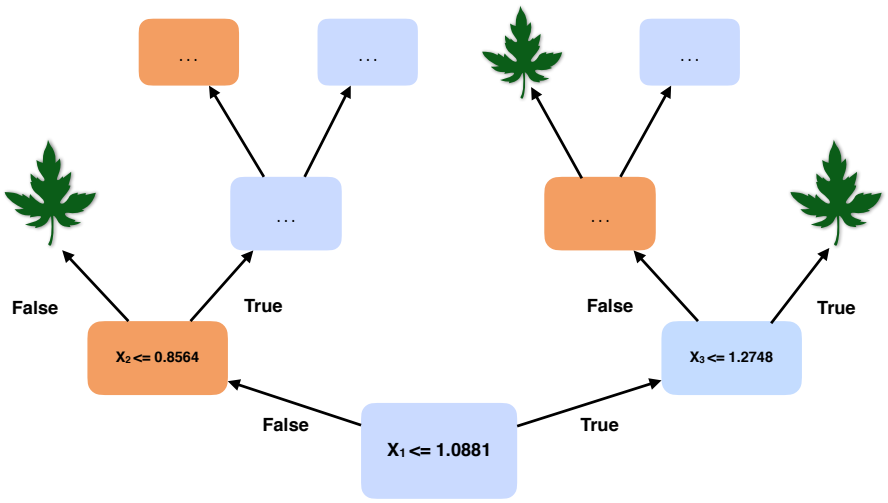


FIG. 4. An illustration of a decision tree, with nodes filled by decision rules inferred from the features of the training data. Inference is made on unseen samples by following the decision rules until they reach a leaf node, where no more splits are being made and the algorithm makes its prediction.

between the samples. An illustration of a (shallow) decision tree is shown in Fig. 4. During inference, samples at a node will be split into left and right nodes according to their value for the feature X_i. This process is repeated for each node of the tree until one reaches a *leaf node*, where no more splits are made. The final leaf node returns the final prediction for all samples that end up in such leaf. In a classification task, this is the probability of belonging to each class which comes from the fraction of training samples in each class at that leaf node. In regression, the final prediction is a single value of the (continuous) target variable given by the average value from the training samples that end up in that leaf node.

Training a decision tree is equivalent to selecting decision rules which optimally partition the training data. This requires adopting a metric to define the best split. Different metrics exist to choose the best feature and the best value for that feature at any given node of the tree. Typically, the best decision rule is that maximizing the decrease in the error in the predictions. Mathematically, this is defined as follows. Consider a node n of a decision tree, where the split made by feature X divides N_n samples into two subsets of N_{n_L} and N_{n_L} samples in the children nodes n_R and n_L, respectively. The splitting feature X is chosen such that it maximizes the impurity decrease $\Delta p(n)$, defined as

$$\Delta p(n) = p(n) - \frac{N_{n_L}}{N_n} p(n_L) - \frac{N_{n_R}}{N_n} p(n_R), \tag{12}$$

where $p(n)$, $p(n_L)$ and $p(n_R)$ are the impurity measures at node n, n_L and n_R, respectively. For a classification problem, the impurity p can be the Gini index,

$$p_G(n) = 1 - \sum_{j=1}^{c} s_j(n)^2, \tag{13}$$

or the Shannon entropy (or, information gain),

$$p_E(n) = -\sum_{j=1}^{c} s_j(n) \log_2 s_j(n), \tag{14}$$

where $s_j(n)$ is the proportion of samples that belong to class j at node n and c is the total number of classes. Both the Gini index and entropy are zero when the node is pure i.e., when all samples belong to one class, and are maximum when the samples are evenly distributed in classes. For regression, the impurity p is usually given by the mean squared error or the mean absolute error. Training a decision tree is therefore not done via gradient descent but rather via a greedy 'top-down'

approach that analyzes all predictors and all splitting values for each predictor to choose the optimal set of predictors and splitting values that will have the least sum of squared error.

Decision trees are often referred to as a "white-box" model, as it is simple to understand their inner working and and to interpret their predictions. This simplicity comes at the expense of accuracy; they typically create over-complex trees which provide an excellent fit to the training data, but do not generalize to new, independent data. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree can partially help to mitigate this problem but are often not enough to fully avoid it. In addition to this, small variations in the training data lead to completely different tree being generated, leading to a model with high variance error in the predictions.

A better solution comes from combining a large number of individual decision trees into ensemble learners. In the following sections, I will introduce ensembles of decision trees and two specific types of ensembles used in our work; *random forests* and *gradient boosted trees.*

A. Ensembles of decision trees

Since individual trees generally over-fit the training data, they are often combined together to form a more robust ensemble estimator. The two main approaches to combine decision trees are *bagging* [14] and *boosting* [15]. The two approaches form ensembles that differ substantially in the trade-off between the models' ability to minimize bias and variance in the predictions. Bagging estimators are effective at decreasing variance, but have no effect on the bias; trees learn independently on bootstrapped training samples and the final prediction of the ensemble is given by the average over individual trees' predictions. On the other hand, boosting can reduce both the bias and the variance contributions to the error in the predictions [16] by aggregating trees iteratively, in such a way that subsequent trees learn to correct the mistakes of the previous ones.

1. Random Forests

Random forests are an ensemble of decision trees which combine trees by means of bagging [17]. They are found to outperform most other popular machine learning algorithms (as for example Naive Bayes, Support Vector Machines and k-nearest neighbours) for many classification problems [18]. Ensemble methods usually work by combining predictions of several base estimators in order to improve generalisability over a single estimator [19, 20]. Random forests are formed by an ensemble of decision trees, each acting in parallel on the data and equally contributing to the final prediction. The prediction of the ensemble is given by the average of the predictions from individual trees. In the case of classification, this is the probability of belonging to each individual class, whereas for regression it is a single value for the target variable.

The power of random forests to reduce variance only manifests when randomness is introduced in order to reduce correlations between the classifiers within the ensemble. The main elements of randomness in a random forest are twofold. The first is that each tree of the forest is trained on a random subset of samples drawn with replacement from the original training set. This is a common feature amongst all bagging estimators. The second is that the best feature at a node of a single tree is chosen not out of all features, but out of a sub-set of randomly drawn features. Using feature bagging reduces correlations between decision trees that can arise when only a few features are strongly predictive of the final output. A third element of randomness can be introduced by splitting features randomly rather than using the criteria described above; these are known as extremely randomized forests. Although individual trees become even weaker estimators, the ensemble predictive power can be increased as it reduces correlations between trees even further. In general, a random forest can slightly increase the bias (with respect to the bias of a single non-random tree) but its variance dramatically decreases due to averaging over the trees; the latter usually more than compensates for the increase in bias, hence yielding an overall better model.

Similar to decision trees, random forests also have hyperparameters that need to be optimized using cross-validation methods. These include all hyperparameters of decision trees, as well as the number of trees in the forest and the fraction of features to randomly draw at each node when looking for the best split.

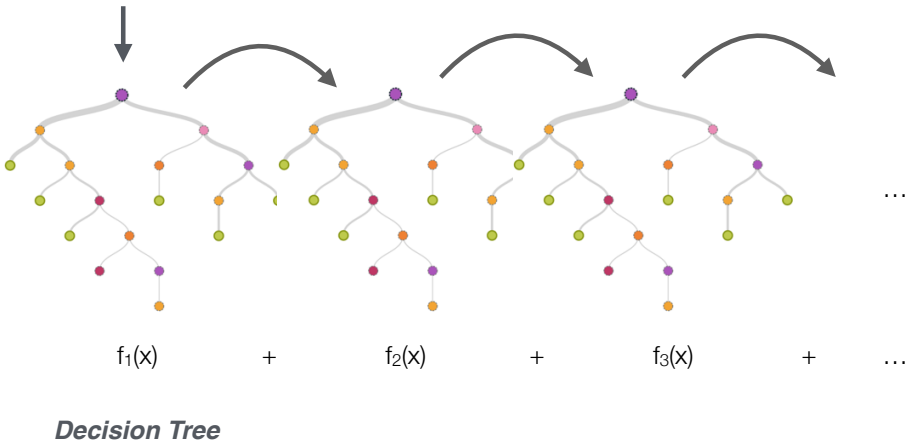


FIG. 5. An illustration of a gradient boosted tree, where new decision trees are iteratively added to the existing ensemble following a gradient-descent optimization procedure.

2. *Gradient boosted trees*

Gradient boosted trees are a boosting ensemble of decision trees [21–23]. The main difference with random forests is that trees are added one at a time to the ensemble, such that each new tree acts to correct errors made by the existing ensemble. This is in contrast to bagging, where the contribution of all predictors is weighted equally in the bagged ensemble. The basic idea of gradient boosted trees is to combine the idea of boosting and gradient descent optimization to construct the ensemble. The performance of gradient boosted trees can be expressed in terms of the loss function; the aim of the algorithm is to minimize the loss evaluated for the training data by adopting a gradient-descent optimization procedure. At each step, the algorithm computes the gradient of the loss function with respect to the predicted value of the ensemble and adds trees that move the loss in the direction of the gradient. In practice, this requires a clever way of mapping gradients to decision trees.

Mathematically, this can be described as follows. At any given iteration m in the gradient boosted tree, a new decision tree $f_m(\mathbf{x})$ is added to the existing ensemble $F_{m-1}(\mathbf{x})$ such that the prediction for a given training sample i, $F_m(\mathbf{x}_i)$, is updated as

$$F_m(\mathbf{x}_i) = F_{m-1}(\mathbf{x}_i) + f_m(\mathbf{x}_i), \tag{15}$$

where $\mathbf{x}_i$ is the input vector for that training sample. An illustration of the iterative addition of decision tree learners in order to construct a gradient boosted tree is shown in Fig. 5. The accuracy of the gradient boosted tree is quantified by the loss function, measuring how well the model's learnt parameters fit the data. The aim is to build a sequence of M trees which minimizes the loss function between the target value y and the predicted one $\hat{y} = F_M(\mathbf{x})$. Gradient boosted trees solve this minimization problem using gradient-descent optimization. The parameters of a decision tree, consisting of both the decision rules and the target variable for that tree, are chosen to point in the direction of the negative gradient of the loss function with respect to the ensemble's predictions. As an example, consider a regression task with the loss function to be the mean squared error between the target value y and the prediction $\hat{y}$. At iteration m, the loss function L is given by the mean squared error between the target value y and the current prediction $\hat{y} = F_{m-1}(\mathbf{x})$ for N training samples,

$$L(y, F_{m-1}) = \sum_i^N \frac{(y_i - F_{m-1}(\mathbf{x}_i))^2}{2}. \tag{16}$$

The negative gradient of the loss function with respect to the predictive model for each training sample i is given by

$$r_i = -\frac{\partial L(y, F_{m-1})}{\partial F_{m-1}}\bigg|_i = y_i - F_{m-1}\left(\mathbf{x}_i\right). \tag{17}$$

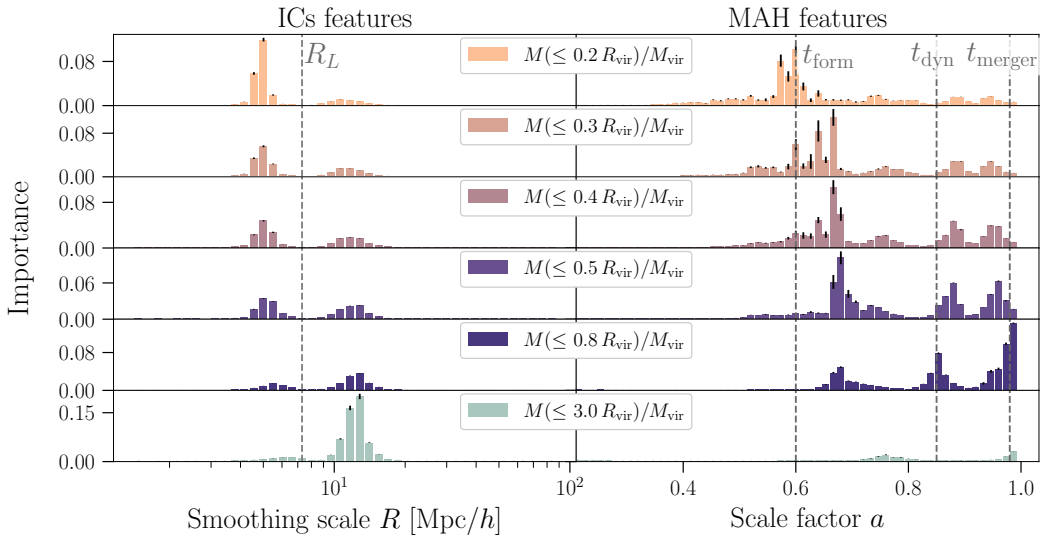


FIG. 6. Example of feature importances in the context of predicting the mass profiles of galaxy clusters. We show how important are the initial density field smoothed on various spatial scales (left) and the halo mass as a function of time (right) in predicting the mass profile of clusters from the inner region (top panel) to the outskirts (bottom panel).

Therefore, when choosing the mean squared error as the loss function, the decision tree at iteration m is trained to predict the residuals r of the current predictions with respect to the true target values. This procedure is repeated until adding further trees does not yield further changes in the loss. The final prediction is the sum of the predictions of all the models. Gradient boosted trees are flexible enough to minimize any loss function, as long as it is differentiable. Gradient boosting and gradient descent share similarities in that both aim to minimize the loss, but do so by different means: gradient descent "descends" the gradient by introducing changes to the model parameters, whereas gradient boosting descends the gradient by iteratively fitting a new model to the residual errors of the previous model.

B. Interpreting ensembles of decision trees via feature importances

Ensembles of decision trees provide an excellent trade-off between interpretability and accuracy. In addition to the predictive power of ensembles of trees, they also allow for interpretability of their learning procedure. This can be achieved using a metric known as *feature importances* [24] to measure the relevance of each input feature in training the algorithm to predict the correct output. This is a crucial aspect of our machine learning application; it will allow us to extract physical knowledge from the machine learning results.

The importance of the j-th feature X_j from a single tree t of the ensemble is given by

$$\mathrm{Imp}_t\left(X_j\right) = \sum_{n\in\{n\,\text{is split on feature}\,X_j\}} \frac{N_n}{N_t}\left[p - \frac{N_{n_R}}{N_n}p_R - \frac{N_{n_L}}{N_n}p_L\right], \tag{18}$$

where N_t, N_n, N_{n_R}, N_{n_L} are the total number of samples in the tree t, at the node n, at the right-child node n_R and at the left-child node n_L, respectively. The sum in the equation is over all n nodes where the feature X_j makes the split. The impurity p is given by the choice of splitting criterion, which in our case is the mean squared error. The final importance of feature X_j given by the ensemble of T trees is the normalized sum over the importances from all trees,

$$\mathrm{Imp}(X_j) = \frac{\sum_{t=1}^{T}\mathrm{Imp}_t\left(X_j\right)}{\sum_{j=1}^{J}\mathrm{Imp}\left(X_j\right)}. \tag{19}$$

An example of the use of feature importances in the context of cosmological structure formation is shown in Fig. 6. A gradient boosted tree model was trained to predict the mass profile of galaxy clusters given as input a variety of features: (1) the initial density field smoothed on different spatial scales, and (2) the mass of the cluster as a function of time during its assembly history. Fig. 6 shows the importance of each of these features in predicting the final mass profile.

IV. DEEP LEARNING ALGORITHMS

Neural networks are a machine learning model inspired by the way biological neural networks process information in the human brain [25]. Deep neural networks have a long history [26] but the notion of *deep learning* was introduced in the 2000s when large artificial neural networks were used for Boltzmann machines [27, 28]. The word *deep* refers to the hierarchical structure in neural networks, where many stacks (or, layers) of neurons are placed between the inputs and the outputs [29]. The outputs of the first layer become the inputs of the second layer, and so on through each layer in the network until the final output (citeDengLearning. The output from each layer is distinctly different and more abstract than the original input data; this level of increasing abstraction introduced by each layer makes deep learning algorithms more difficult to understand than standard machine learning algorithms.

Deep learning networks quickly demonstrated their success compared to shallow machine learning algorithms. The first examples of this came from two deep learning implementations, the first being AlexNet [30], which reduced the error on the ImageNet Large Scale Visual Recognition Challenge by 12%, and the second ResNet, which also achieved dramatic improvement over existing methods with an error of 3.57% [31]. Since then, deep neural networks have become the standard technique of many image and speech recognition tasks (see e.g. Mehta *et al.* [32] for a review). We first give an overview of neural networks and then describe in detail the workings of convolutional neural networks [33].

A. Neural networks

The fundamental unit of a feed-forward neural network is a neuron, which takes scalar inputs, performs first a linear transformation and then a non-linear one, and finally outputs a real number. The linear transformation takes the form of a dot product with a set of neuron-specific weights and an additional bias, as

$$h = \sigma(\mathbf{W}\mathbf{x} + b), \tag{20}$$

where h is the neuron output, $\mathbf{x}$ is the input, $\mathbf{W}$ and b are a multiplicative weight matrix and an additive bias respectively, and $\sigma(\cdot)$ is a non-linear function. Historically, common choices of non-linearities included step-functions (perceptrons), sigmoids and the hyperbolic tangent. More recently, it has become more common to use rectified linear units (ReLUs), leaky rectified linear units (leaky ReLUs), and exponential linear units (ELUs), as they are found to outperform other choices in a variety of tasks [34]. The output $h(\mathbf{x})$ serves as input to the next layer, and so on, until one reaches the output layer. Typically, a neural network is organized by stacking a number of "hidden" layers in between the input and output layers, thus forming a *deep* neural network as shown in Fig. 7. Each hidden layer is in turn made up of a large number of neurons; the power of neural networks stems from how neurons are connected to each other. Since all neurons of neighbouring layers are connected to each other, the layers are also known fully-connected layers.

In summary, the deep neural network can be thought of as a complicated non-linear transformation of the inputs $\mathbf{x}$ into an output y that depends on the weights and biases of all the neurons in the input, hidden, and output layers. This defines a parametric non-linear funtion $f_\Theta(\mathbf{x})$ where Θ are the set of weights and biases of all layers. The predictive power of neural networks expresses itself in the universal approximation theorem, which states that a neural network with a single hidden layer can approximate any continuous, multi-input/multi-output function with arbitrary accuracy [25]. However, the more complicated a function, the more hidden units (and free parameters) are needed to approximate it (at fixed accuracy). Hence, the applicability of the approximation theorem to practical situations is limited by computational power, training data availability and other factors.

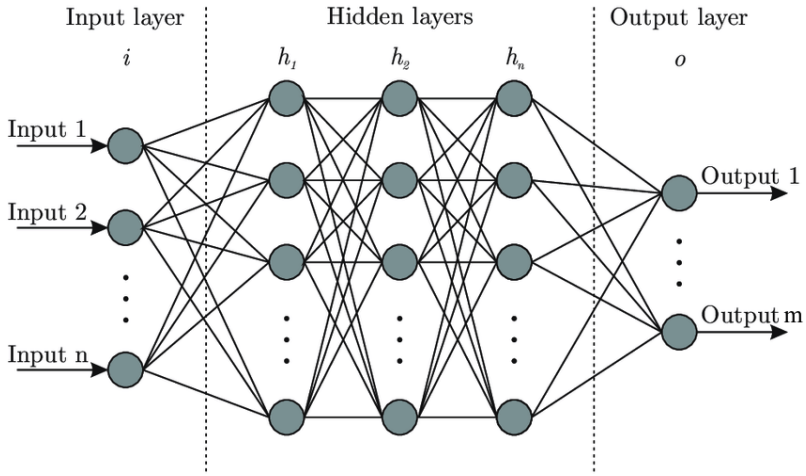


FIG. 7. An illustration of a deep neural network with an input layer, three hidden layers and an output layer.

The basic principles of training a neural network are somewhat similar to those of gradient boosted trees; one specifies a loss function and then updates the parameters, in this case the weights and biases, using gradient descent optimization. First, information is propagated through the network in a feed-forward fashion i.e., forward layer-by-layer through the network from the inputs to the outputs. Unlike gradient boosted trees, the optimization of the parameters requires a more complicated procedure, known as *backpropagation* [35], due to the large number of weights and biases in the different hidden layers and the complexity of deep neural networks. At its core, backpropagation is simply the ordinary chain rule for partial differentiation applied to solve the gradient of the loss with respect to the weights and biases. The term backpropagation comes from the fact that in computing the gradients one moves back from the outputs to each hidden layer, until reaching the input layer. Forward and backward passes of the neural network is typically done many times, where each pass is known as an *epoch*. At each epoch, new updates to the weights are being made in the direction of the negative gradient of the loss with respect to the weights. The convergence of the neural network can be tested by tracking the loss function as a function of number of epochs; once the loss does not change with increasing number of epochs, then the algorithm has reached convergence.

Performance can be improved for particular types of data by making use of inductive biases. Images respond to convolutional neural networks, which are translational invariant and whose pixel topology is incoporated by constructions. If the data consists of time series, then recurrent networks incorporate by construction temporal information and variable length. Finally, if the data is graph-structured, graph neural networks incorporate graph topology in such a way that can be viewed as a generalization of convolutional neural networks beyond gridded images. One should therefore pick different neural architectures depending on the different types of data at hand. Most likely, there exists a neural network on the market that can accommodate your data structure (avoid reinventing the wheel); additionally, complete architectures can be compositions of these layers (and others).

B. Deep convolutional neural networks

Neural networks fail to exploit spatial structure in input data such as images. A $n \times n$ image must be reshaped into a one-dimensional vector of size n^2 in order to train neural networks, which therefore neglects spatial information about the image and requires an enormous amount of weights for every input. This issue required the design of a new class of neural network architectures, namely convolutional neural networks (CNNs), that account for locality in the input data [33, 36]. Convolutional networks are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data,which can be thought of as a 2-D grid of pixels. They are defined as a feed-forward neural network with the use convolution in place of

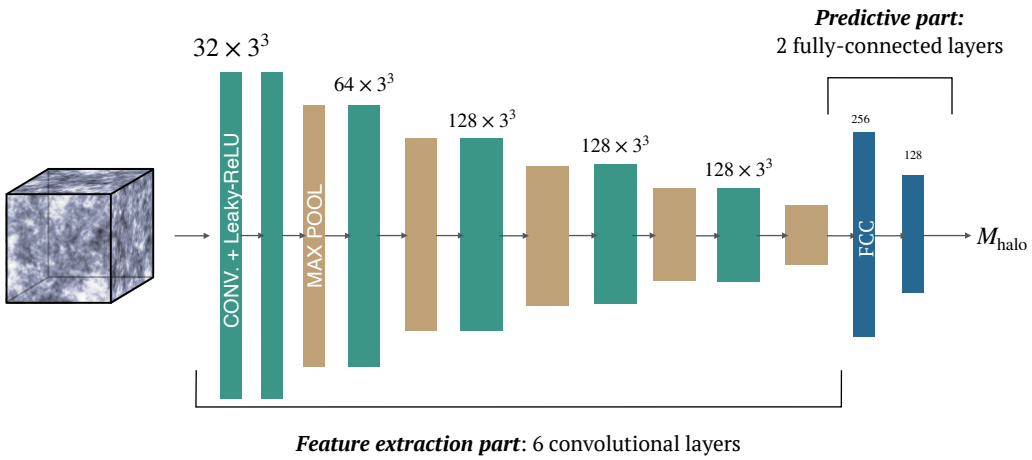


FIG. 8. An example of a convolutional neural network made of six convolutional layers which perform the feature extraction, and two fully connected layers which output the final prediction. In this example, the 3D density field in the initial conditions of a cosmological simulation are mapped to the final mass of the dark matter halo that eventually forms in that initial region of space.

general matrix multiplication in at least one of their layers

CNNs are one of the most powerful techniques at present, yielding breakthrough results in image recognition [30, 31, 37, 38], natural language processing ([39–41]), object detection [42, 43], reinforcement learning, and many other fields.

1. *CNN architecture*

Choosing the exact network architecture is often problem-specific, thus requiring extensive numerical experimentation and intuition. More complicated architectures may be better at capturing complex correlations in the data and learning relevant patterns, but may also lead to overfitting by fitting spurious patterns of the training data. An example of a typical CNN architecture is shown in Fig. 8. The majority of CNN-based architectures have two main components: a feature extraction part and a predictive part. Feature extraction consists of a series of convolutional layers, in which the algorithm learns to extract relevant features from the input data. This process is hierarchical: the first layers learn local, low-level features, which are then combined by subsequent layers into more global, higher-level features. In the predictive part of the CNN, the features are combined to return the final prediction using fully-connected layers. We will describe below each of the basic layers involved in most CNN architectures.

Note that there have also been numerous works that move beyond the simple deep, feed-forward neural network architectures, which for example incorporate "skip connections" that allow information to directly propagate to a hidden or output layer, bypassing intermediate ones [31]. By adding skip connections, the network avoids training for the layers that are not useful and that do not add value in overall accuracy. This idea is particularly helpful in cases where adding more layers to a deep learning model leads to a higher training error.

Convolutional layer: In the context of a convolutional neural network, a convolution is a linear operation that involves a dot product between a set of weights and the input summed by a bias term. Although traditionally designed to work on images, CNNs are not just limited to 2-D images but can be generalized to volumetric data, or any n-dimensional data, simply by adopting three-dimensional, or n-dimensional, convolutional kernels. For simplicity, I will describe convolutions assuming the inputs are two-dimensional images but the same arguments apply to higher-dimensional data.

In its most general form, a discrete convolution is defined as the following operation,

$$s(t) = (x * w)(a) \sum_{a=-\infty}^{\infty} x(a)w(t-a)da. \tag{21}$$

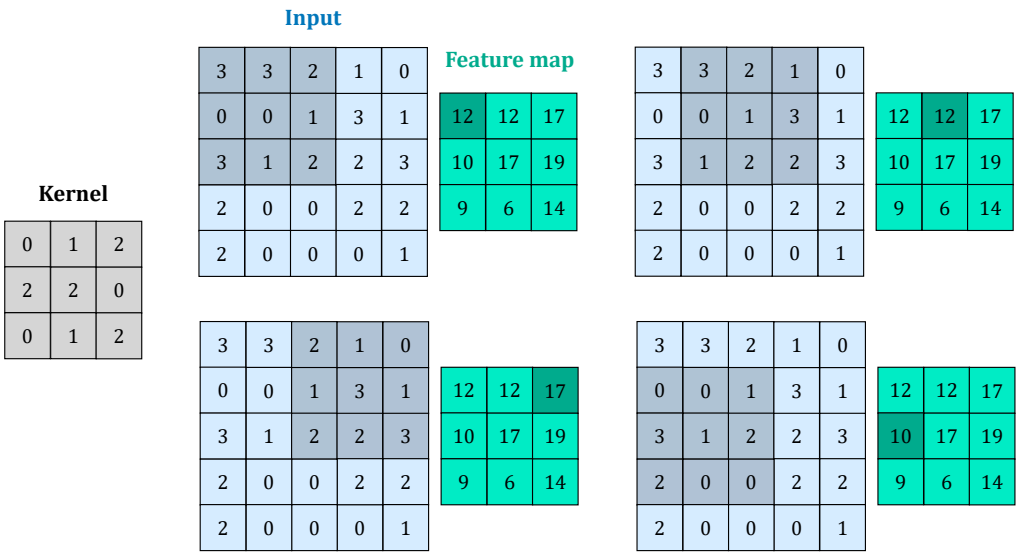


FIG. 9. An example of a convolution between a 3×3 kernel and a 5×5 image to produce the feature map in green. The same kernel is applied to different 3×3 parts of the image, sliding top to bottom and left to right.

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the input, the second argument (in this example, the function w) as the kernel or the filter, and the output s is the feature map. The values of the kernel are learnt by the CNN during the training process. CNNs are often used for 2D images but can be generalized to volumetric data, or any n-dimensional data, simply by adopting three-dimensional, or n-dimensional, convolutional kernels. For example, a two-dimensional image I is convolved with a two-dimensional kernel K such that

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{22}$$

Technically speaking, many neural network libraries implement a related function called the cross-correlation, which is the same as a convolution but without flipping the kernel, i.e. $S(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$. In the context of machine learning, the learning algorithm will learn the appropriate values of the kernel and so it is irrelevant whether or not one adopts convolution or cross-correlation.

The size k of the $k \times k$ filter is usually much smaller than the size of the input data, where typical values of k are 3, 5, and 7. Crucially, the weights of the filter remain the same as the filter is applied to different parts of the input image. Therefore, if a given filter is designed to detect a specific type of feature in the input, it will discover that feature anywhere in the image. This capability is commonly referred to as *translation equivariance* and is a powerful property when interested in whether a certain feature is present, independent of where it is located in the input. This is one of the main reasons why convolutional neural networks are particularly suited for images. The output from a single multiplication of the filter with the input is a single value, but as the filter is applied to different patches of the image, the end result is a two-dimensional matrix called a *feature map*. Each pixel in the feature map indicates the strength of the detected feature in different regions of the input image. An example of a 3×3 kernel that is convolved with a 5×5 image to produce the feature map in green is shown in Fig. 9.

The size of the resulting feature map is controlled by three hyperparameters that have to be set prior to training: the number of convolutional filters, the stride, and amount of zero-padding. The number of convolutional filters determines the number of features to be learnt at any given layer. The stride is the number of pixels by which to slide the filter across the image when performing the convolution. For example, if the stride is one, the filter is moved one pixel at a time across the image, whereas if the stride is two, the filter slides over two pixels at a time. With zero-padding, one pads the image with zeros around the border in order to center the filter on elements at the edge of the image. With appropriate zero padding around the image, and sliding the convolution filter with a stride of 1, the output map will be of the same size as the input. Instead, one can decide

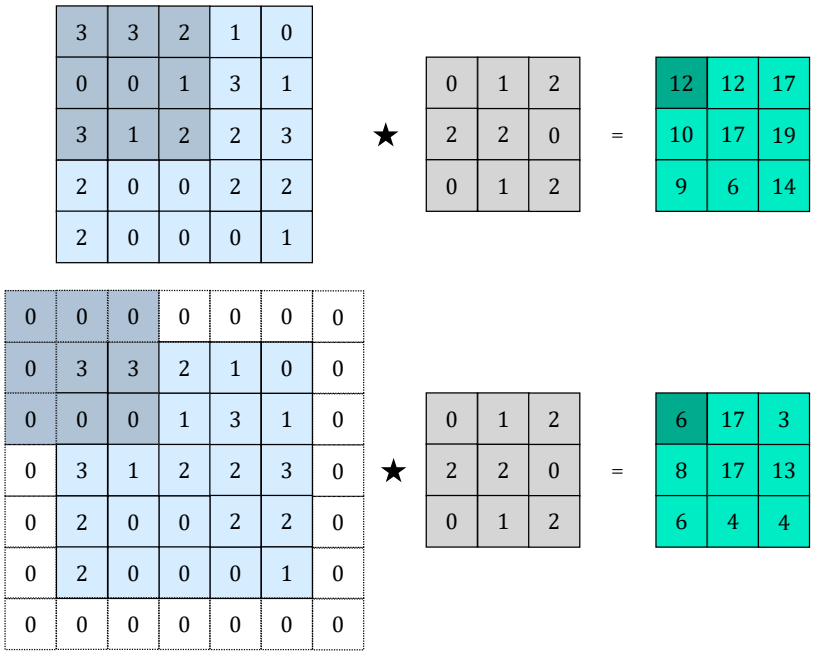


FIG. 10. Top: A convolution with stride one and no zero-padding; the kernel is never centred at the locations of the pixels at the boundary of the image. Bottom: a convolution with stride 2 and additional zero-padding so that each image pixel (include those at the boundaries) are centred with the kernel.

to reduce the size of the output map by two by choosing a stride of 2. Fig. 10 shows an example of a convolution with stride 1 and no padding (top panel) and a convolution with stride 2 and zero-padding (bottom panel). The output size is given by $n_{\text{out}} = (n_{\text{in}} + 2p - k + 1)/s$, where n_{out}, n_{in}, k are the size of the output feature map, the input and the kernel respectively, p is the amount of zero-padding and s is the stride. A detailed review of the arithmetic of convolutional layers can be found in Dumoulin and Visin [44].

Typically, a convolutional layer is composed of several feature maps, each constructed using a different convolutional filter, so that multiple features can be extracted from the image in a single convolutional layer. Similar to fully connected layers, each value in the feature maps is passed through a non-linear activation function, as for example a ReLu [45]. Moreover, the input image may be composed of several channel (as for example RGB channels for images) so that the kernel size must also adapt to the channel size of the inputs. A more complete example of a single convolutional layer with a multi-channel input, several kernels and a non-linear activation function is shown in Fig. 11.

One advantageous property of CNNs is *translational equivariance*. The translation equivariance is obtained by means of the convolutional layers. In fact, if the input image is translated to the right by a certain amount, the feature maps generated by convolutional layers are shifted by the same amount and direction. Mathematically, if we apply a translation $T_{\mathbf{v}}(\mathbf{x}) = \mathbf{x} + \mathbf{v}$ to an image $\mathbf{I}$ and then perform a convolution f, the resulting response image $f(T_{\mathbf{v}}(\mathbf{I}))$ will be the same as if we first performed convolution to $\mathbf{I}$ and then applied the same translation $T_{\mathbf{v}}$ to the response $f(\mathbf{I})$.

Pooling layer: Convolutional layers are often followed by pooling layers, which reduce the dimensionality of a feature map by taking the average (*average-pooling*) or the maximum value (*max-pooling*) in small, usually 2×2, regions of the feature maps. They act separately on each feature map, meaning that from a set of N feature maps when obtains the same number of N pooled maps. The effect of the pooling layer is to produce lower-resolution feature maps, which are less sensitive to small changes in the position of the feature in the image compared to the higher-resolution feature maps returned by the convolutional layer.

Pooling layers also offer the additional property of *translational invariance*. Since pooling extracts the representative

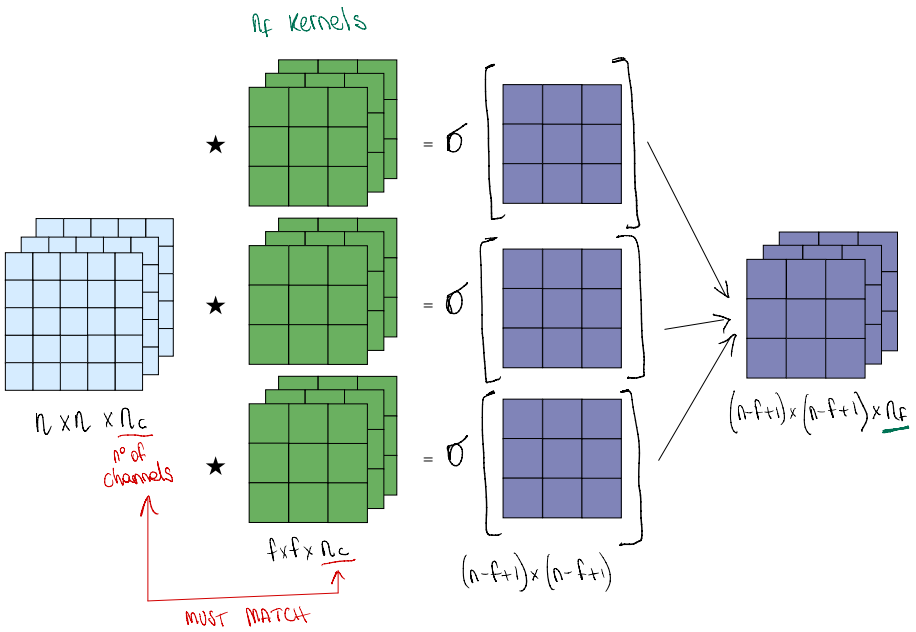


FIG. 11. An example of one layer of a convolutional network which include multi-channel input, several convolutional kernels and non-linear activation function.

statistics (max or average) from the feature map, the location of such statistic in the original feature map is irrelevant. Note that CNNs are not naturally equivariant and invariant to rotation, scaling, and affine transformations. Hence, data augmentation techniques must be used to make CNNs more robust to such geometric transformations.

Batch normalization layer: It is sometimes common to add a *batch-normalization layer* [46], which normalizes the inputs of a batch by first subtracting the batch mean and dividing by the batch standard deviation and then rescaling and shifting the normalized values using two parameters γ and β, which are learnt during backpropagation. This layer is usually placed in between the convolutional layer and the non-linear activation function. Its success is usually attributed to the fact that it reduces the negative impact of changes in the distribution of layer inputs caused by updates to the network parameters in the preceding layers. This effect is known as *internal covariate shift*. However, the exact reasons for its effectiveness are still a matter of debate [47].

C. Training

Similar to regular deep neural networks, convolutional neural networks are trained for a number of epochs, each consisting of a forward pass, where the input passes through the network and reaches the output layer, and a backward pass, where gradients are backpropagated and all the weights in the convolutional layers are updated according to the negative gradient of the loss function. The updates are usually suppressed by multiplying the gradients by a small number α, known as the *learning rate*, which takes values between 0 and 1. The learning rate controls how quickly the model descends towards the minimum of the loss and is one of the most important hyperparameters in the network. If the learning rate is set too low, training will progress very slowly towards the minimum, as only tiny updates are made to the weights each time. If the learning rate is set too high, it can cause drastic changes to the weights, leading to divergent behaviour in the loss function.

The training data can be divided into one or more subsets, called *batches*, which are forward- and backward- propagated through the network independently. In this way, weights are updated after each batch. The size of the batches is a hyperparameter which must be set prior to training. If all training samples are contained in a single batch rather than sub-divided into

multiple batches, the learning algorithm is called *batch gradient descent*. If instead each batch consists of a single sample or a subset of the training data, the learning algorithm is called *stochastic gradient descent* or *mini-batch gradient descent*, respectively. One epoch is therefore made of N forward and backward passes for each of the N batches.

Deep convolutional neural networks contain a large number of hyperparameters to be set before training, making their tuning a challenging task. These involve architecture-specific parameters, such as the number of layers (including convolutional, pooling, batch-normalization and fully-connected layers), the number of epochs, the gradient descent optimizer, the learning rate and the choice of loss function, as well as layer-specific parameters. For convolutional layers, these include the size of the kernels, the choice of non-linear activation function, the amount of stride and zero-padding; for pooling layers, the amount of down-sampling and the type of pooling (max or average); for fully-connected layers, the number of neurons and the choice of non-linearity. The large number of hyperparameters to tune in convolutional neural networks makes a fully grid-based optimization search infeasible. Most of these choices require a large number of numerical trial-and-error stages, which can be done in a systematic way to explore the sensitivity of the learning to the various parameters and the degeneracies between the parameters.

1. Regularization

One issue when training deep neural networks is that they are generally over-parametrized. It has in fact been demonstrated that there exists a major redundancy in the parameters used by a deep neural network [48]. This means that when minimizing the negative log-likelihood with a deep neural network model, one almost always encounters the problem of overfitting. The algorithm tends to fit the samples of the training data $\mathcal{D}$ extremely well but fails to learn patterns that are generalizable to unseen data. To overcome this issue, one modifies the loss function of the neural network in such a way that prevents the algorithm from overfitting and improves its generalizability. This is known as regularization.

One way to introduce regularization is to adopt priors over the weights. Following Bayes' theorem, the goal of the neural network then becomes to maximize the posterior distribution $p\left(\boldsymbol{w} \mid \mathcal{D}\right) = p\left(\mathcal{D} \mid \boldsymbol{w}\right) p\left(\boldsymbol{w}\right)$, rather than the likelihood $p\left(\mathcal{D} \mid \boldsymbol{w}\right)$. The loss function, $\mathcal{L}$, is then given by

$$\mathcal{L} = -\ln\left[p(\boldsymbol{w} \mid \mathcal{D})\right] = -\ln\left[p\left(\mathcal{D} \mid \boldsymbol{w}\right)\right] - \ln\left[p(\boldsymbol{w})\right], \tag{23}$$

where the first is the likelihood term, or predictive term $\mathcal{L}_{\text{pred}}$, and the second is the prior term, or regularization term $\mathcal{L}_{\text{reg}}$. If $\boldsymbol{w}$ are given a Gaussian prior, this yields L2 regularization i.e. $p(\boldsymbol{w}) = \lambda \sum_i w_i^2$. This prior penalizes large values for the weights as it penalizes the sum of the squared values of the weights, thus leading to more generalizable solutions. If $\boldsymbol{w}$ are given a Laplacian prior, then one obtains L1 regularization i.e. $p(\boldsymbol{w}) = \lambda \sum_i |w|$. In this case, the prior also penalizes large values for the weights but this time by penalizing the sum of the absolute values of the weights. The choice of Laplacian prior has the additional benefit that it induces sparsity on the weights by driving most weights to be zero. A Laplacian prior thus combines the idea of model compression and regularization in that it simultaneously (i) improvex the optimization during training by preventing overfitting and (ii) compresses the neural network model into the least number of parameters without loss in performance. Gaussian priors are typically adopted for the weights of the convolutional kernels, while Laplacian prior are more typically applied to the weights of the fully-connected layers. The regularization parameter λ weighs the prior term relative to the likelihood term in the loss function; its value sets the balance between an overly-complex model (which overfits and has a high variance) and an overly-simple model (which underfits and has a high bias).The advantage of these form of regularization is that it can be incorporated in terms of priors on the weights.

There exist many other regularization techniques, including for example dropout where a fraction of randomly-selected nodes of a fully-connected later (or kernels in a convolutional layer) are 'dropped out' or set to zero at every training iteration. The nodes are dropped by a dropout probability of p which must be set a priori. My personal preference is to not adopt these forms of regularization, as they do not have a direct Bayesian interpretation.

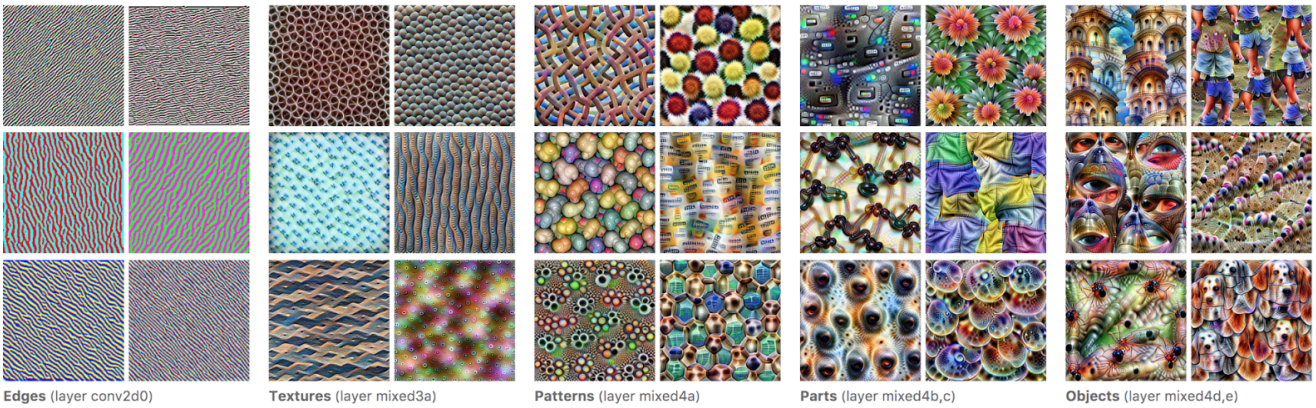


FIG. 12. Feature maps visualization showing how GoogLeNet, trained on the ImageNet dataset, builds up its understanding of images over many layers. The CNN first learns low-level, local features such as edges, which are then combined by subsequent layers of the network into more global, higher-level features such as objects. Visualization from [50].

D. Representation learning

One important and powerful aspect of deep learning algorithms lies in its ability to learn relevant features from the raw data, a task known as *representation learning*. This is in contrast to most of the other machine learning algorithms, such as random forests or gradient boosted trees, which require pre-processing the data into a selected set of features used for training. The hierarchical structure of deep learning models is thought to be crucial to their ability to represent complex features. For example, in convolutional neural networks, the first layers learn local low-level features, as for example edges in images, which are then combined by subsequent layers of the network into more global, higher-level features ([49]). This is because each pixel in a convolutional layer is only a function of the $k \times k$ pixels in the previous layer that are contained inside the $k \times k$ kernel of the convolutional filter. Since typically $k \in \{3, 5, 7\}$, the algorithm will only learn local features. As more convolutional layers are stacked on top of each other, the region of the input that any given pixel is a function of increases. The size of this region at any specific layer is called *receptive field* of the layer. The receptive field increases layer-by-layer making each layer sensitive to features at increasingly larger scales. In this way, both local and global information propagate through the network. An example of the features extracted from the different convolutional layers of a CNN is shown in Fig. 12.

V. INTERPRETABILITY IN DEEP NEURAL NETWORKS

Going back to the accuracy vs. interpretability trade-off, deep CNNs provide the most striking example of extremely high accuracy and low interpretability. Model interpretability is currently lacking for CNNs (e.g. [51]); there is very little insight into the internal operation of these complex models, or how they achieve such good performance. The feature maps generated by each convolutional layer in the network, show patterns in the data which are hard to relate to human-interpretable quantities. In addition to this, the complex network of inter-connected neurons of the CNN also makes it difficult to quantify how individual features then map onto the resulting predictions. Most efforts related to interpretability have focused on understanding (i) the functionality of the different hidden layers, (ii) the relationship and interconnection between neurons and (iii) how features are assembled throughout the network up to the final prediction ([52]).

To this end, the machine learning community has tried to answer such questions by developing tools based on feature visualization ([50, 53, 54]), attribution ([55–58]), and dimensionality reduction ([59]). Feature visualization tools usually project a model's learnt feature map back to the pixel space using a technique called deconvolution. The aim is to provide insight into what types of features deep neural networks are learning at specific layers in the model. However, due to the large number of feature maps in the network, these methods are usually only applied to few, low-level feature maps in the first layers

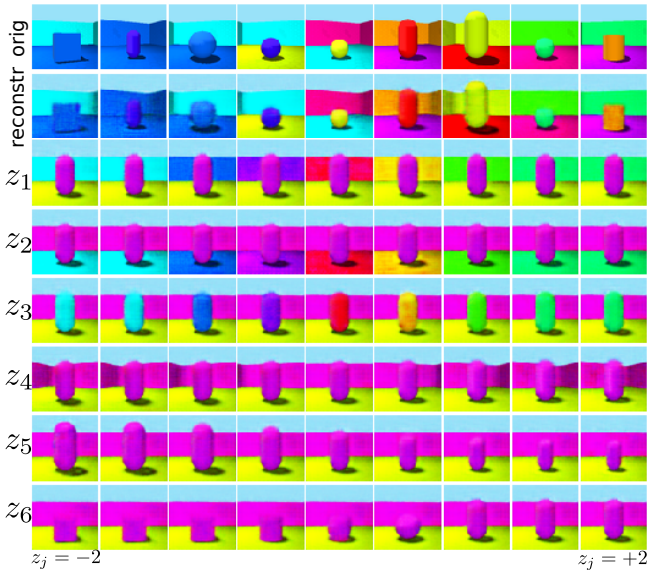


FIG. 13. First row: original images. Second row: reconstructed images by the deep learning model. Remaining rows: reconstructed images when each latent variable is systematically varied, while keeping all other latent variables fixed.

and so can only provide limited insight. Moreover, qualitative evaluation of such maps can be difficult to turn into quantitative statements about the learnt features. The most common approach for attribution techniques is a *saliency map*, a colormap that highlights pixels of the input image that most caused the output classification ([56, 57]). These methods are also limited in that they do not take into account correlations between pixels, or that they only display pixels of the input that are relevant to a single output. Finally, dimensionality reduction methods such as t-SNE have been used (not only for neural networks) to reduce the dimensionality of high-dimensional features into lower dimensions, while trying to preserve the characteristics of the data.

More recently, the field of interpretability and explainability has made progress through *disentangled representation learning* (DLR) [60]. Existing end-to-end black-box deep learning models directly learn hierarchical representations of the object, which fail to extract the hidden attributes carried in representations with human-like semantics and generalization ability. Instead of extracting hierarchical features through a series of subsequent layers, DRL is a learning paradigm which aims to obtain low-dimensional representations which identify the underlying generative factors (or factors of variation) in the observed data. This low-dimensional representation lies in a manifold often called 'latent space'. In other words, the original data is *compressed* into a lower-dimensional representation in latent space. With DRL, the goal is obtain representations of the observed data that carry semantic meanings. In addition to identifying the underlying generative factors of the data, DRL also typically aims to *disentangle* those underlying generative factors into distinct latent variables. In other words, a disentangled representation should separate the distinct, independent and informative generative factors of variation in the data. In this way, single latent variables are sensitive to changes in single underlying generative factors, while being relatively invariant to changes in other factors. The ability to separate independent factors of variation into separate independent latents is known as *disentanglement*.

A frequently used dataset in DRL is Shape3D [61], which contains 3D objects with six distinct factors of variation, i.e., floor colour, wall colour, object colour, scale, shape and orientation. DRL aims at separating these factors and encoding them into independent and distinct latent variables in the representation space. In this case, the latent variables controlling object shape will change only with the variation of object shape and be constant over other factors. Analogously, it is the same for variables controlling other factors including size, color and so on. An example of this is shown in Fig. 13: the first and second rows show the original and reconstructed images by the neural network. Indeed in this case, the network is asked to generate images that resemble the original ones of the training data; in doing so, the network compressed the input images into

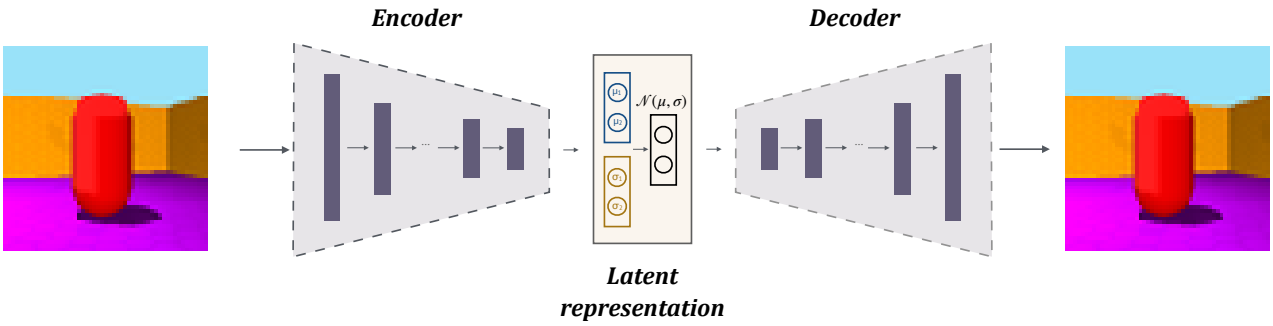


FIG. 14. A variational autoencoder is an unsupervised, deep generative model that aims to generate images that resemble those in the training data. In doing so, it generates a disentangled latent representation that contains the underlying generative factors of the data.

a low-dimensional disentangled latent space. The subsequent rows show image reconstructions generated by systematically varying the latent space, while all other latents are kept fixed. Clearly, latent z_1 is controlling wall colour, z_2 floor colour, and so on. The opposite of a disentangled representation is as expected, an entangled representation. An entangled representation identifies latent factors that each map to more than one aspect of the generative process. In the Shape3D example above, an entangled latent factor may explain the wall colour, shape and object colour in a single dimension. This latent factor would certainly explain some variance in the data, but by convolving many aspects of the generative process together, it would be difficult to understand which object properties were causing which aspects of this variation. Therefore, disentanglement allows for a clean interpretation of the latent variables in terms of the independent generative factors, or degrees of freedom, in the data.

A. Disentangled Representation Learning with variational autoencoders

The question then becomes how can one turn a CNN 'black-box' architecture into one that yields a disentangled representation of the data. The most successful deep learning models that have done so are variational auto-encoders (VAEs), a class of probabilistic deep generative models. The goal of a generative model is to learn an implicit distribution $\mathcal{P}$ from which the training set $X = \{x_0, x_1, ..., x_n\}$ is drawn. This typically involves building a parametric model $\mathcal{P}_\theta$ that tries to be as close as possible to $\mathcal{P}$. Once $\mathcal{P}_\theta$ is obtained, one can typically sample from such distribution to generate new data or even evaluate the likelihood $p_\theta(\mathbf{x})$. There exists many different types of generative models, ranging from deep belief networks [27], Variational AutoEncoder [62], Generative Adversarial Networks [63] and Wasserstein GAN [64] and normalizing flows [65]. Here, we will focus on variational autoencoders as they are the most used in the context of DLR.

As their name suggests, VAEs are a type of autoencoder [62, 66]. Autoencoders are latent variable models, meaning that they introduce 'latent variables' to explain the observed data. There are two components in an autoencoder: the first maps the input data into a lower-dimensional vector $\mathbf{z}$, and the second which reconstructs something that that resembles the input data starting given $\mathbf{z}$ [67, 68].. The former part of the algorithm is known as the *encoder* and is typically parametrized by a neural network; the second is the *decoder* which is also parametrized by a neural network. The latent representation encodes the relevant information contained in the input data into the lowest possible dimensions. The network architecture for the encoder and the decoder varies between simple feed-forward neural networks or convolutional neural networks depending on the use case. An illustration of a VAE architecture is shown in Fig. 14.

The main difference between a standard autoencoder and a variational one is that the former case is deterministic while the latter is probabilistic: in the former case, every input maps to a low dimensional vector $\mathbf{z}$, while in the latter case each input maps to a distribution over the possible values of z. The fundamental limitation of a deterministic autoencoder is that the latent

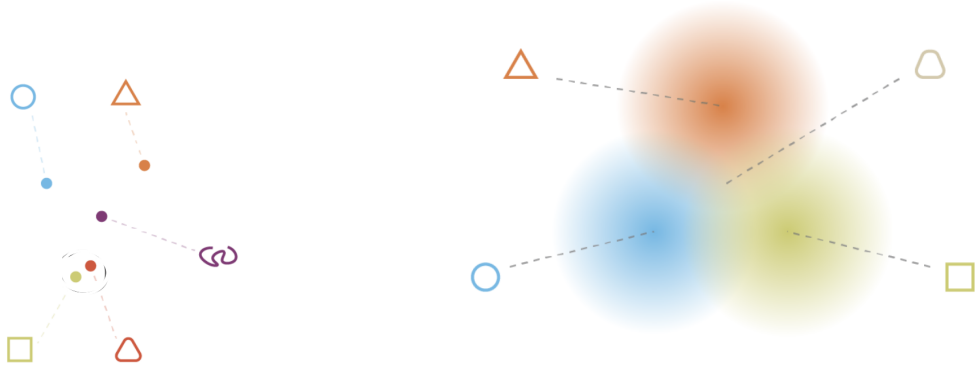


FIG. 15. Latent space of an autoencoder (left panel) and that of a variational autoencoder (right panel). Autoencoders map each input to a single point in latent space, yielding a discontinuous latent space which returns unrealistic outputs when sampling from a previously unseen region in latent space. Variational autoencoders map each input to a distribution over possible latent values, thus yielding a continuous latent space which allows for better interpolation.

space they convert their inputs to may not be continuous, or allow easy interpolation. If the latent space has discontinuities, the decoder will return unrealistic outputs when sampling from any region that overlaps a discontinuity in latent space. This is because during training, the decoder was never given examples of encoded vectors coming from that region of latent space. On the other hand, VAEs yield continuous latent representation. Instead of encoding an input as a single point, VAEs encode it as a distribution over the latent space, as shown in Fig. 15. In this way, the auto-encoder learns the probability distribution functions over latent space, $p(\mathbf{z}|\mathbf{x})$, where $\mathbf{z}$ and $\mathbf{x}$ are the latent and input variables respectively, resulting in a smooth latent space which can be easily interpolated. The decoder on the other hand learns the generative process $p(\mathbf{x}|\mathbf{z})$.

The property of disentanglement in VAEs is imposed via the loss function which we now formally derive. In variational autoencoders, one assumes that the observations $\mathbf{x}$ are generated following a random process involving unobserved latent variables $\mathbf{z}$ according to some parametric distribution $p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})$, where θ are the parameters of this distribution which we aim to adjust so that the marginal distribution $p_\theta(\mathbf{x})$ matches closely the empirical distribution of the data. A typical choice is to assume a Gaussian likelihood for the data $p_\theta(x|z) = \mathcal{N}(x; f_\theta(z), \sigma^2)$, so that the network is responsible for predicting the maximum likelihood estimate. Training the neural network involves finding the optimal parameters $\theta^\star$ that maximize the marginal likelihood of the model:

$$\theta^\star = \arg\max_\theta p_\theta(\mathbf{x}) = \arg\max_\theta \int p_\theta(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) \mathrm{d}\mathbf{z}. \tag{24}$$

However, this kind of computation is often intractable and requires the use of approximation techniques such as variational inference which allow you to efficiently train the model and find the optimal $\theta^\star$ parameters. The idea behind variational inference is to introduce a family of parametric distribution $q_\phi(\mathbf{z}|\mathbf{x})$ which aims to model the true posterior $p_\theta(\mathbf{z}|\mathbf{x})$. In other words, the $q_\phi(\mathbf{z}|\mathbf{x})$ model returns an estimate of the true posterior density $p_\theta(\mathbf{z}|\mathbf{x})$ for each example $\mathbf{x}$. This means that we want to minimize the distance between the two distributions $q_\phi(\mathbf{z}|\mathbf{x})$ and $p_\theta(\mathbf{z}|\mathbf{x})$, which can be quantified by the Kullback-Leibler (KL) divergence $D_{\mathrm{KL}}(q_\phi(z|x)\,||\,p_\theta(z|x))$.[3] The optimization problem is then

$$\theta^\star, \phi^\star = \arg\min_{\theta,\phi} D_{\mathrm{KL}}(q_\phi(z|x)\,||\,p_\theta(z|x)). \tag{25}$$

[3] The KL divergence is a measure rooted in information theory of the difference between two probability distributions. In general, the KL divergence of distribution Q from P, $D_{\mathrm{KL}}(P||Q))$, describes the loss of information when Q is used to approximate the reference distribution P. This is not a symmetric function, as the information content in Q about P is not equivalent to information content in P about Q. It is a non-negative quantity and takes the value $D_{\mathrm{KL}}(P||Q)) = 0$ if and only if the two distributions are identical.

The KL divergence can be expanded as follows,

$$D_{\mathrm{KL}}(q_\phi(z|x)\,||\,p_\theta(z|x)) = \int q(z|x)\log\frac{q(z|x)}{p(z|x)}\mathrm{d}z \tag{26}$$
$$= \mathbb{E}_q[\log q(z|x) - \log p(z|x)] \tag{27}$$
$$= \mathbb{E}_q[\log q(z|x) - \log p(z,x) + \log p(x)] \tag{28}$$
$$= \mathbb{E}_q[\log q(z|x) - \log p(z) - \log p(x|z)] + \log p(x) \tag{29}$$
$$= D_{\mathrm{KL}}(q(z|x)||p(z)) + \log p(x) - \mathbb{E}_q[\log p(x|z)] \tag{30}$$

Reordering the terms of this expression leads to:

$$\log p(x) = \mathbb{E}_q[\log p(x|z)] - D_{\mathrm{KL}}(q(z|x)||p(z)) + D_{\mathrm{KL}}(q_\phi(z|x)\,||\,p_\theta(z|x)). \tag{31}$$

Taking into account the fact that the KL divergence (hence the last terms above) is always positive, this leads to the following lower bound on the marginal log likelihood of $\mathbf{x}$, known as the Evidence Lower Bound (ELBO):

$$\log p(x) \geq \mathbb{E}_q[\log p(x|z)] - D_{\mathrm{KL}}(q(z|x)||p(z)). \tag{32}$$

Contrary to the original marginal likelihood, the ELBO is now completely tractable, as neither $p(x)$ or $p(z|x)$ appear in the RHS. Finally, the loss function that is minimized when training a VAE is given by

$$\mathcal{L}_{\mathrm{VAE}} = \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] - \beta\, D_{\mathrm{KL}}(q_\phi(z|x)\,||\,p(z)) \tag{33}$$

where β is a hyperparameter that is introduced to balance the two terms in the loss function [69]. The first term, $\mathbb{E}_{q_\phi(z|x)}\left[\log p(\mathbf{x}|\mathbf{z})\right]$, is the usual log likelihood of the data we observed in the measurement space $\mathbf{x}$ given the latent space $\mathbf{z}$. This is known as the reconstruction term, which measures the performance of the encoding-decoding scheme. The second term is a regularization term, which measures how different the learned latent distribution $q_\phi(z|x)$ is from a prior we have on the latent distribution $p(z)$. The loss function reflects a trade-off between maximising the likelihood and staying close to the prior distribution. This trade-off is natural for Bayesian inference problem and express the balance that needs to be found between the confidence we have in the data and the confidence we have in the prior. β serves as the weighting term that regulates this trade-off.

Note that we have some liberty to choose some structure for the latent variables. A typical choice is to assume Gaussian representations for the latent prior $p(z)$ and the approximate posterior $q_\phi(z|x)$, such that $p(z) \sim \mathcal{N}(\mathbf{0}, \mathbb{I})$ and $q_\phi(z|x) = \mathcal{N}(\mu_\phi(\mathbf{x}), \sigma^2_\phi(\mathbf{x}))$. This means that the encoder outputs two vectors: a vector of means $\boldsymbol{\mu}$ and a vector of standard deviations $\boldsymbol{\sigma}$ for each input $\mathbf{x}$. In this case, the KL divergence term (for a single i-th sample) takes the closed form,

$$\mathcal{D}_{\mathrm{KL}}(\mathcal{N}(\mu,\sigma);\mathcal{N}(\mathbf{0},\mathbb{I})) = -\frac{1}{2}\sum_{i=1}^{L}\left[1 + 2\log\sigma_i - \mu_i^2 - \sigma_i^2\right]. \tag{34}$$

where L is the dimensionality of the latent space. The KL divergence term can also be interpreted as a regularization term in machine learning language. Without the regularization term, the encoder can generate very different means $\boldsymbol{\mu}$ and small standard deviations $\boldsymbol{\sigma}$ as there are no limits to what values μ and σ can take. This would yield latent variables which are tightly clustered and far apart from each other, making interpolation difficult. Instead, the latent variables should be as close as possible to each other (while still being distinct), allowing smooth interpolation. The regularization term is introduced in the loss function to ensure that the latent space satisfies continuity i.e., two points close in latent space should return similar content once decoded, and completeness i.e., all points sampled in latent space should return meaningful content once decoded.

The role of the KL term in the loss function is to promote independence between the latents. This encourages the model to find a disentangled latent space, where independent factors of variation in the density profiles are captured by different, independent latents. Here, independence is intended in terms of both linearly and non-linearly uncorrelated variables. Hence, linear correlation measures such as the Pearson correlation coefficient are insufficient. Compressing the information

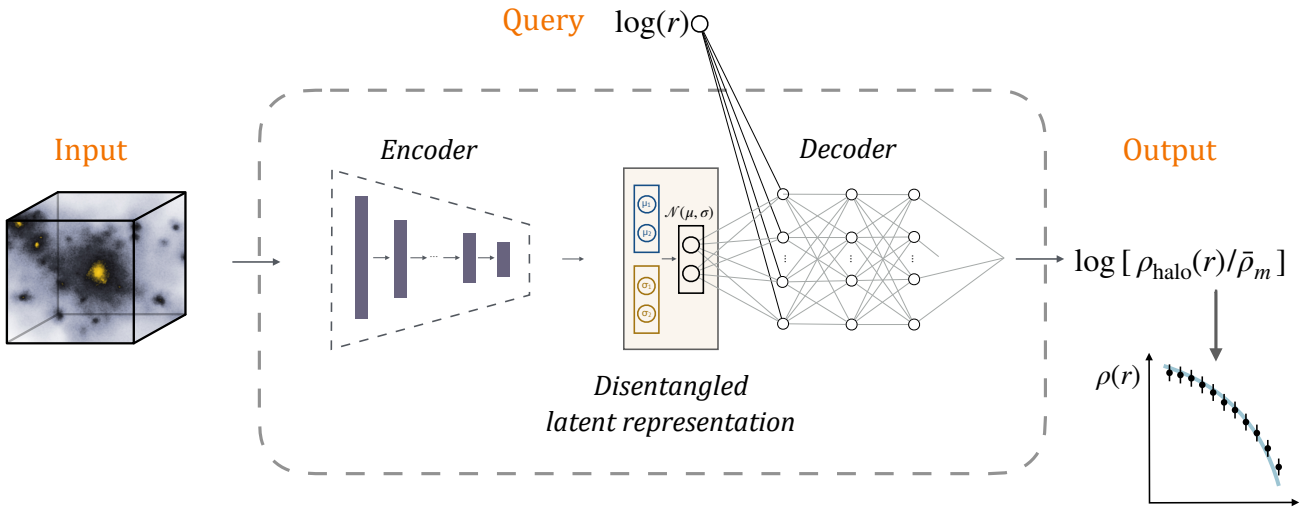


FIG. 16. The interpretable variational encoder (IVE) consists of an encoder compressing the 3D density field of a cosmological simulation containing each halo into a low-dimensional latent representation, followed by a decoder mapping the latent representation and a given value of r to the spherically-averaged density $\rho(r)$. In this illustration, the latent space is two-dimensional; however, the dimensionality of the latent space can be increased to any arbitrary value. The latent representation only retains the information required by the model to predict the halo density profiles, allowing us to interpret the representation as independent factors of variability in the density profiles.

in the input data into a disentangled low-dimensional latent representation can be thought of as a non-linear principal component analysis (PCA). PCA is a dimensionality-reduction technique to linear transform a set of correlated variables into linearly-uncorrelated components. The components describe linearly-uncorrelated factors of variability of the data set and dimensionality reduction is achieved by discarding components which describe negligible variability of the data. The encoder plays the role of a non-linear PCA: it performs a *non-linear* transformation of the input data into *disentangled* components. The degree of disentanglement of the latent space therefore crucially affects our ability to interpret the latent space in terms of independent factors of variability in the density profiles.

B. Disentangled Representation Learning with supervised models

Variational autoencoders are unsupervised learning models which are primarily used as a generative model. Many applications in physics however involve supervised problems, where one wishes to infer a new output $\mathbf{y}$ from a set of inputs $\mathbf{x}$. An interpretable variational encoder (IVE) is supervised encoder-decoder architecture that replaces the decoder in a VAE with a neural network $p_\theta(y|\mathbf{z}(\mathbf{x}))$ that predicts an output y given the latents $\mathbf{z}$ encoding information about the input $\mathbf{x}$. The encoder has a similar role to that of the VAE: it compresses the information in the input into a compact low-dimensional latent representation. The decoder takes the latent representation and an additional input, known as the *query* q, and predicts an output $y(q)$. The query is the argument of the function y we aim to learn. Unlike in unsupervised variational autoencoders, the latent representation need not describe the input data completely; it only needs to capture the information necessary to predict the output of interest. This is achieved through the query: it induces the latent space to retain the information used by the decoder to predict $y(q)$ for any given value of q, by construction. The presence of the query therefore plays a crucial role in enabling interpretability. Without it, the information about the output would be spread throughout the parameters of the whole encoder and decoder, as it is for a standard feed forward neural network.

This network architecture was first developed by *SciNet* [70] where it demonstrated its ability to rediscover the known parameters in various 1D toy examples. It was then further expanded in [71], where we consider a real-world scenario which deals with 3D inputs and where the physically relevant quantities are not known a priori. An example of the IVE architecture

applied to a problem in cosmological structure formation is shown in Fig. 16. In this example, the IVE was trained to generate a set of disentangled latent parameters that can model the density profile of a dark matter halo, starting from the raw 3D density field in the cosmological simulation around the centre of the halo. The IVE architecture is both *interpretable* and *explainable*. Interpretability is achieved by compress the relevant information to predict the output of interest in a disentangled latent representation. Explainability denotes the ability to map the interpretations onto existing knowledge in the relevant science domain; this is achieved by evaluating the *mutual information* between the latent variables and relevant physical quantities regarding the halos' density profiles. This directly reveals the information content of each latent variable in relation to the halos' density profiles.

VI. CONCLUSIONS

The trade-off between interpretability and predictive performance in machine learning poses a challenge for physicists who employ machine learning tools in their science. Our goal as physicists is to understand nature by employing mathematical models, and thus we are not satisfied with models that produce outputs which we cannot interpret or explain. Interpretability serves the dual purpose of trusting the results of machine learning models, and understanding complex, non-linear physical processes which cannot be accessed with traditional statistical techniques. Efforts in turning "black-box" deep learning models into explainable ones may enable new data-driven scientific discoveries.

[1] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, Learning from data (2012).

[2] T. J. Hastie, R. Tibshirani, and J. H. Friedman, The elements of statistical learning: Data mining, inference, and prediction, 2nd edition, in *Springer Series in Statistics* (2005).

[3] D. J. C. MacKay, Information theory, inference, and learning algorithms, IEEE Transactions on Information Theory **50**, 2544 (2003).

[4] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)* (Springer-Verlag, Berlin, Heidelberg, 2006).

[5] K. P. Murphy, *Machine learning: a probabilistic perspective* (Cambridge, MA, 2012).

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, Mastering the game of go with deep neural networks and tree search, Nature **529**, 484 (2016).

[7] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu, Feature selection: A data perspective, CoRR **abs/1601.07996** (2016).

[8] J. Duchi, E. Hazan, and Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, J. Mach. Learn. Res. **12**, 2121–2159 (2011).

[9] D. Kingma and J. Ba, Adam: A method for stochastic optimization, International Conference on Learning Representations (2014).

[10] S. Geman, E. Bienenstock, and R. Doursat, Neural networks and the bias/variance dilemma, Neural Computation **4**, 1 (1992).

[11] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, Classification and regression trees (1984).

[12] J. R. Quinlan, Induction of decision trees, Mach. Learn. **1**, 81 (1986).

[13] S. L. Salzberg, C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993, Machine Learning **16**, 235 (1994).

[14] L. Breiman, Bagging predictors, Machine learning **24**, 123 (1996).

[15] Y. Freund and R. E. Schapire, Game theory, on-line prediction and boosting, in *Proceedings of the Ninth Annual Conference on Computational Learning Theory*, COLT '96 (ACM, New York, NY, USA, 1996) pp. 325–332.

[16] R. E. Schapire, Y. Freund, P. Bartlett, and W. S. Lee, Boosting the margin: a new explanation for the effectiveness of voting methods, Ann. Statist. **26**, 1651 (1998).

[17] L. Breiman, Random forests, Machine Learning **45**, 5 (2001).

[18] R. Caruana and A. Niculescu-Mizil, An empirical comparison of supervised learning algorithms, in *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06 (ACM, New York, NY, USA, 2006) pp. 161–168.

[19] T. G. Dietterich, Ensemble methods in machine learning, in *Proceedings of the First International Workshop on Multiple Classifier Systems*, MCS '00 (Springer-Verlag, London, UK, UK, 2000) pp. 1–15.

[20] T. G. Dietterich, An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization, Mach. Learn. **40**, 139 (2000).

[21] Y. Freund and R. E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, Journal of computer and system sciences **55**, 119 (1997).

[22] J. H. Friedman, Greedy function approximation: A gradient boosting machine., Ann. Statist. **29**, 1189 (2001).

[23] J. H. Friedman, Stochastic gradient boosting, Computational Statistics & Data Analysis **38**, 367 (2002), nonlinear Methods and Data Mining.

[24] G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts, Understanding variable importances in forests of randomized trees, in *Advances in Neural Information Processing Systems 26*, edited by C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Curran Associates, Inc., 2013) pp. 431–439.

[25] M. A. Nielsen, *Neural networks and deep learning*, Vol. 25 (Determination press San Francisco, CA, USA:, 2015).

[26] C. M. Bishop, *Neural Networks for Pattern Recognition* (Oxford University Press, Inc., New York, NY, USA, 1995).

[27] G. E. Hinton, S. Osindero, and Y.-W. Teh, A fast learning algorithm for deep belief nets, Neural Comput. **18**, 1527 (2006).

[28] R. Salakhutdinov and G. Hinton, Deep boltzmann machines, in *Artificial intelligence and statistics* (2009) pp. 448–455.

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning* (2016).

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, Imagenet classification with deep convolutional neural networks, in *Advances in Neural Information Processing Systems 25*, edited by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Curran Associates, Inc., 2012) pp. 1097–1105.

[31] K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) , 770 (2015).

[32] P. Mehta, M. Bukov, C.-H. Wang, A. r. G. R. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, A high-bias, low-variance introduction to Machine Learning for physicists, Phys. Rep. **810**, 1 (2019), arXiv:1803.08823 [physics.comp-ph].

[33] Y. Lecun and Y. Bengio, Convolutional networks for images, speech, and time-series, in *The handbook of brain theory and neural networks*, edited by M. Arbib (MIT Press, 1995).

[34] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, Activation functions: Comparison of trends in practice and research for deep learning. arxiv 2018, arXiv preprint arXiv:1811.03378 (2018).

[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, Nature **323**, 533 (1986).

[36] Y. LeCun, Y. Bengio, and G. Hinton, Deep learning, Nature **521**, 436 (2015).

[37] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, Going deeper with convolutions, 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) , 1 (2014).

[38] K. Simonyan and A. Zisserman, Very deep convolutional networks for large-scale image recognition, CoRR **abs/1409.1556** (2014).

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, arXiv e-prints , arXiv:1810.04805 (2018), arXiv:1810.04805 [cs.CL].

[40] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, Exploring the Limits of Language Modeling, arXiv e-prints , arXiv:1602.02410 (2016), arXiv:1602.02410 [cs.CL].

[41] K. Clark, M.-T. Luong, C. D. Manning, and Q. Le, Semi-supervised sequence modeling with cross-view training, in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (Association for Computational Linguistics, Brussels, Belgium, 2018) pp. 1914–1925.

[42] W. Ouyang, X. Zeng, X. Wang, S. Qiu, P. Luo, Y. Tian, H. Li, S. Yang, Z. Wang, H. Li, *et al.*, Deepid-net: Object detection with deformable part based convolutional neural networks, IEEE Transactions on Pattern Analysis and Machine Intelligence **39**, 1320 (2016).

[43] A. Diba, V. Sharma, A. Mohammad Pazandeh, H. Pirsiavash, and L. Van Gool, Weakly supervised cascaded convolutional networks (2017).

[44] V. Dumoulin and F. Visin, A guide to convolution arithmetic for deep learning, arXiv e-prints , arXiv:1603.07285 (2016), arXiv:1603.07285 [stat.ML].

[45] V. Nair and G. E. Hinton, Rectified linear units improve restricted boltzmann machines, in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10 (Omnipress, USA, 2010) pp. 807–814.

[46] S. Ioffe and C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15 (JMLR.org, 2015) pp. 448–456.

[47] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, How does batch normalization help optimization?, in *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18 (Curran Associates Inc., USA, 2018) pp. 2488–2498.

[48] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, Predicting parameters in deep learning, Advances in neural information processing systems **26** (2013).

[49] Q. V. Le, M. Ranzato, R. Monga, M. Devin, G. S. Corrado, K. Chen, J. Dean, and A. Y. Ng, Building high-level features using large scale unsupervised learning, 2013 IEEE International Conference on Acoustics, Speech and Signal Processing , 8595 (2011).

[50] C. Olah, A. Mordvintsev, and L. Schubert, Feature visualization, Distill 10.23915/distill.00007 (2017), https://distill.pub/2017/feature-visualization.

[51] Q.-s. Zhang and S.-c. Zhu, Visual interpretability for deep learning: a survey, Frontiers of Information Technology & Electronic Engineering **19**, 27 (2018).

[52] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev, The building blocks of interpretability, Distill (2018).

[53] M. D. Zeiler and R. Fergus, Visualizing and understanding convolutional networks, in *Computer Vision – ECCV 2014*, edited by D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars (Springer International Publishing, Cham, 2014) pp. 818–833.

[54] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, Striving for simplicity: The all convolutional net, CoRR **abs/1412.6806** (2014).

[55] K. Simonyan, A. Vedaldi, and A. Zisserman, Deep inside convolutional networks: Visualising image classification models and saliency maps, CoRR **abs/1312.6034** (2013).

[56] B. Zhou, A. Khosla, À. Lapedriza, A. Oliva, and A. Torralba, Learning deep features for discriminative localization, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) , 2921 (2015).

[57] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, Grad-cam: Visual explanations from deep networks via gradient-based localization, 2017 IEEE International Conference on Computer Vision (ICCV) , 618 (2016).

[58] R. Fong and A. Vedaldi, Interpretable explanations of black boxes by meaningful perturbation (2017) pp. 3449–3457.

[59] L. van der Maaten and G. Hinton, Visualizing data using t-SNE, Journal of Machine Learning Research **9**, 2579 (2008).

[60] X. Wang, H. Chen, S. Tang, Z. Wu, and W. Zhu, Disentangled representation learning (2023), arXiv:2211.11695 [cs.LG].

[61] H. Kim and A. Mnih, Disentangling by factorising, in *Proceedings of the 35th International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 80, edited by J. Dy and A. Krause (PMLR, 2018) pp. 2649–2658.

[62] D. Kingma and M. Welling, Auto-encoding variational bayes (2014).

[63] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, Generative Adversarial Networks, arXiv e-prints , arXiv:1406.2661 (2014), arXiv:1406.2661 [stat.ML].

[64] M. Arjovsky, S. Chintala, and L. Bottou, Wasserstein gan (2017), arXiv:1701.07875 [stat.ML].

[65] I. Kobyzev, S. J. Prince, and M. A. Brubaker, Normalizing flows: An introduction and review of current methods, IEEE Transactions on Pattern Analysis and Machine Intelligence **43**, 3964 (2021).

[66] D. J. Rezende, S. Mohamed, and D. Wierstra, Stochastic backpropagation and approximate inference in deep generative models, in *Proceedings of the 31st International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 32, edited by E. P. Xing and T. Jebara (PMLR, Bejing, China, 2014) pp. 1278–1286.

[67] H. Bourlard and Y. Kamp, Auto-association by multilayer perceptrons and singular value decomposition, Biological Cybernetics **59**, 291 (1988).

[68] G. E. Hinton and R. S. Zemel, Autoencoders, minimum description length and helmholtz free energy, in *NIPS* (1993).

[69] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner, beta-vae: Learning basic visual concepts with a constrained variational framework, in *International conference on learning representations* (2016).

[70] R. Iten, T. Metger, H. Wilming, L. del Rio, and R. Renner, Discovering Physical Concepts with Neural Networks, Phys. Rev. Lett. **124**, 010508 (2020).

[71] L. Lucie-Smith, H. V. Peiris, A. Pontzen, B. Nord, J. Thiyagalingam, and D. Piras, Discovering the building blocks of dark matter halo density profiles with neural networks, Phys. Rev. D **105**, 103533 (2022), arXiv:2203.08827 [astro-ph.CO].