Machine learning

Jan Kieseler[a]
[a]KIT, Karlsruhe, Germany

Advanced machine learning techniques have become ubiquitous: from computer vision algorithms found on a plethora of small devices such as cameras or smartphones to the recent rise of tremendously powerful large language models. Also in high energy particle physics, these techniques have become essential and have led to a significant increase in physics reach, from simple feed-forward algorithms used to distinguish signal and background processes to more complex neural networks that utilise the underlying physics structure of the data. This section will cover the basics of neural networks and their training and will then discuss examples of the building blocks that make up modern machine learning algorithms, aiming to provide a tool box for their further application in physics analyses.

These lecture notes give a short overview of the basic principles of machine learning, exclusively focusing on deep learning. They also cover neural network building blocks adapting to the structure of the data that should be processed by the algorithms and discuss examples of applications in high energy physics. There is already a large amount of literature covering these topics and reliable and well curated material on the internet, in particular for the basic principles, which is why the overview here is kept short and aims to give an intuitive understanding of the ideas behind the covered machine learning techniques. A very concise, yet rather comprehensive summary of the basic principles and neural network building blocks can be found for example in Ref. [1], also containing references to the relevant publications. In principle, the material that is worth covering in this context would correspond to at least one full course. Therefore, only a few examples of relevant methods and techniques are discussed here with the intent that they could serve as a stepping stone towards a more in-depth study of the subject or as a starting point for applications of deep neural networks to physics studies. For the latter, knowing in particular what it means to exploit the structure of the data in the architecture of the neural network is useful, which is why this part will receive the most focus.

This chapter should be cited as: Machine learning, Jan Kieseler, DOI: 10.23730/CYRSP-2025-009.189,
in: Proceedings of the 2023 European School of High-Energy Physics,
CERN Yellow Reports: School Proceedings, CERN-2025-009, DOI: 10.23730/CYRSP-2025-009, p.189.

1 Basic principles

The most basic building block of a neural network is a dense neural network layer. The operation a dense layer ϕ performs given the input vector x can be described by a weight matrix ω, a bias vector b and an activation function θ:

$$h = \phi(x) = \theta(\omega\, x + b), \tag{1}$$

where h is the output vector. The parameters of ω as well as b are free parameters and are learned during the neural network training through gradient descent, covered in more detail later, where the learnable parameters are updated such that the difference between the neural network output and the desired *true* output is minimised. The weight matrix ω can change the dimensionality of the output with respect to the input. A deep feed-forward neural network, also referred to as multi-layer perceptron (MLP), can be built by stacking these dense neural network layers, where each layer k passes on its output to the next $(k+1)$, in other words:

$$h^{(k+1)}(h^{(k)}) = \theta(\omega_{k+1} h^{(k)} + b_{k+1}). \tag{2}$$

This equation recursively defines an MLP, also depicted in Figure 1, where the information is passed from an input layer through hidden layers to the output layer. At each layer, an element of the (hidden) representation is also often referred to as *node*. By learning the free parameters of that model, an MLP can act as a universal function approximator.

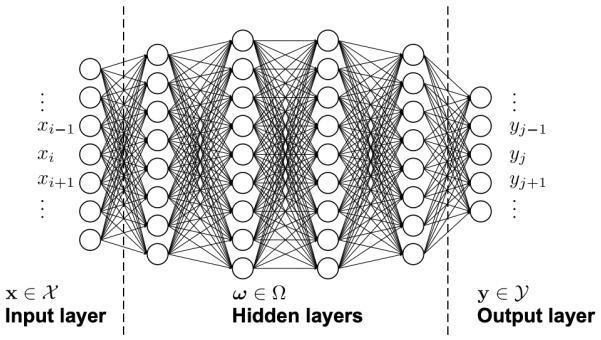

Fig. 1: A sketch of a feed-forward neural network, or multi-layer perceptron. Each node is illustrated by a circle and weights are illustrated by connections. Activation functions and bias vectors are omitted.

An activation function θ appearing in each layer is needed to enable the neural network to describe non-linear dependencies. This can easily be verified if θ is omitted:

$$h^{(k+1)}(h^{(k)}) = \omega_{k+1} h^{(k)} + b_{k+1} = \omega_{k+1} \left(\omega_k h^{(k-1)} + b_k\right) + b_{k+1}. \tag{3}$$

Setting $\omega = \omega_{k+1}\omega_k$ and $b = \omega_{k+1} b_k + b_{k+1}$, the operation of network layer k and $k+1$ can be reduced to one operation. Recursively applied to a neural network of arbitrary depth, this would allow to represent a neural network of arbitrary depth by a single linear operation, limiting the set of functions it can approximate, the expressivity, significantly.

While the only conceptual requirement for the activation function in a hidden layer is that it is non-linear, in practice, there are some restrictions: it should provide numerically stable output and a

numerically stable gradient. As it will be applied many times within the neural network it should also be computationally simple and it should not significantly scale the hidden representations, which could lead to *vanishing* or *exploding* outputs and gradients. Therefore, typical activation functions are applied element wise to the vectors and have a derivative of about one at zero. Two often used examples are tanh, RELU and ELU:

$$\mathrm{RELU}(x) = \max(x, 0) \tag{4}$$

$$\mathrm{ELU}(x) = \begin{array}{ll} e^x - 1 & 0 > x \\ x & 0 \leq x \end{array} \tag{5}$$

In most cases, the exact choice of the hidden-layer activation functions does not have a large impact on the final result. A more comprehensive survey can be found e.g. in Ref. [2].

1.1 Training

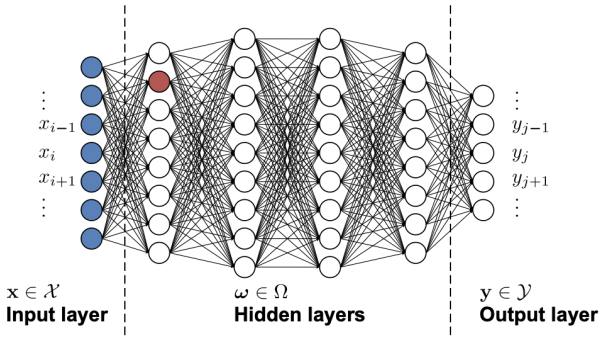

Fig. 2: Activation of a single node in the neural network.

Keeping variance of the inputs, the hidden layers, and the outputs within reasonable constraints is useful for a fast convergence of the training process. This is very similar to practical considerations when performing fits of a functional form to data points (e.g. discussed in Chapter **Practical Statistics**), where also well chosen start parameters as well as reasonably defined parameter ranges help the fit to converge. Therefore, also the input data x is typically normalised, such that the values have approximately a mean of zero and a variance of one. Assuming N inputs that are uncorrelated and normal distributed, the distribution at the red node in Figure 2 would also correspond to a normal distribution, with a variance of N. To avoid that the variance increases in a similar manner with each layer, the weights ω_1 of the first hidden layer can be initialised also following a normal distribution, but scaled by $1/\sqrt{N}$, such that the variance remains one. This choice is referred to as Glorot initialisation [3]. Since also the activation function is applied before feeding the output to the next layer, it is often beneficial to chose initialisation and activation functions that preserve this property, at least to a good approximation. For example, Glorot can be paired with a tanh activation function, while ELU and RELU can be paired with He initialisation [4].

To train the parameters of the neural network, a cost (or loss) function needs to be defined. This function quantifies how well the neural network approximates the desired output, given a set of weights. An example is a classic linear regression, where the parameters a and b of a function $f(x) = ax + b$ are

fit to data points. As discussed in Chapter **Practical Statistics**, this can be done e.g. by using the least square method or through a χ^2 minimisation. In the context of neural networks, the mean of the least squares or the χ^2 define a loss function for fitting the parameters a and b. In both cases, the value of the function $f(x)$ is compared to the true value of y for each point and the (weighted) mean difference is minimised. The training of a neural network follows the exact same principle, but with orders of magnitude more free parameters and typically with a larger set of training points (or data set). One of the most common used loss functions for regression tasks is the mean-squared error (MSE) loss, in this case evaluated for the neural network Φ:

$$\min 1/N \sum_{i}^{N} \left((\Phi(\omega,\, x_i) - y_i)^2 \right) = \min \mathrm{MSE}(\Phi(\omega,\, x), y). \tag{6}$$

Here, x, y, and ω represent all points or all weights respectively. In this case, the neural network should not have a restricted output range, which is why the output layer should have no activation (also referred to as linear activation). In this context, it is important to make the connection to the origin of choosing the MSE loss or a χ^2: the network output is expected to follow a Gaussian likelihood, and therefore minimising the loss function corresponds to minimising the negative log-likelihood that is representative of the problem.

Another prominent task for a neural network is classification, e.g. for distinguishing cat pictures from dog pictures, or signal events from background events. This is a binary classification problem, where the probability for a sample to be identified by the neural network corresponds to a Bernoulli process. With $\hat{y} =: \Phi(\omega, x)$, the probability for a single sample to be identified by the neural network becomes:

$$P(\hat{y}, y) = \hat{y}^y (1 - \hat{y})^{1-y} \tag{7}$$

There, the likelihood for N independent samples factorises as

$$\Pi_{l=1}^{N} (\hat{y}^{(l)})^{y^{(l)}} (1 - \hat{y}^{(l)})^{(1-y^{(l)})} \tag{8}$$

and the negative logarithm becomes:

$$\sum_{l}^{N} \left(y^{(l)} \log(\hat{y}^{(l)}) + (1 - y^{(l)})\, \log(1 - \hat{y}^{(l)}) \right), \tag{9}$$

which is the expression for the binary cross entropy loss used to train binary classification neural networks. Since the probability is strictly defined between zero and one, the activation function for the output layer of the neural network also needs to be bound to the same interval. For this purpose, a sigmoid activation is used for the output layer:

$$\theta_{\text{sigmoid}}(x) = 1/(1 + e^{-x}). \tag{10}$$

In general, this underlines that the choice of the loss function is defined by the output distribution the network is expected to have. In many cases it can be beneficial to either adapt the loss function or redefine the output of the neural network. One example would be if a neural network should be trained to refine

the momentum estimate of a reconstructed jet. Here it can be more beneficial to learn a correction to the unrefined momentum estimate, approximately Gaussian distributed with a mean around one, than to learn to estimate the corrected energy directly, realistically covering an interval that spans two orders of magnitude.

Finally, the training of the neural network is performed using (variants of) gradient descent. Gradient descent is a well established step-wise robust minimisation method, where the parameters of the neural network (or a function) are updated using the gradient of the loss function with respect to the parameters:

$$\omega^{(s+1)} = \omega^{(s)} - \eta \nabla_{\omega^{(s)}} L\left(\Phi(\omega^{(s)},\, x),\, y\right). \tag{11}$$

Here, L is the loss function, s is an update step, $\eta > 0$ is a parameter referred to as learning rate, and x, y, and ω stand for the whole set of training data points or all weights, respectively. The procedure is repeated until a stopping criterion is reached and the loss value does not improve significantly anymore:

$$L\left(\Phi(\omega^{(s)},\, x),\, y\right) - L\left(\Phi(\omega^{(s+1)},\, x),\, y\right) < \epsilon. \tag{12}$$

While gradient descent is only one of many techniques that can be used in principle to minimise the loss function, it is central to all modern deep learning tasks. One of the main reasons is that often the data set used for training, as well as the neural network itself are too large to perform gradient descent on the full data set. In this case, gradient descent naturally extends to stochastic gradient descent on (mini) batches of the data, where only a part of the data set ($\{x\}_s,\, \{y\}_s$) is processed at each step s:

$$\omega^{(s+1)} = \omega^{(s)} - \eta \nabla_{\omega^{(s)}} L\left(\Phi(\omega,\, \{x\}_s),\, \{y\}_s\right) \tag{13}$$

This reduces the computational burden and makes the training of large models feasible. However, it has implications on the stopping criterion and the learning rate as the procedure introduces additional noise into the system. In practice that means that the learning rate needs to be decreased the smaller the batches become, and that the stopping criterion needs to be tuned to account for statistical fluctuations due to the batch size and fluctuations in the gradients from batch to batch. On the other hand, this additional noise can help to find a minimum. Considering that deep neural networks have thousands to billions of parameters, it is not guaranteed that there is always a clear path to one global minimum. More concretely, it is not guaranteed that the optimisation problem is strictly convex. In these situations it has been shown empirically that some additional noise can help to avoid local minima and increases the chance to converge towards a global minimum (or a valley in parameter space that minimises the loss function) [5]. Typically, the algorithms that performs updates of the weights of the neural network are referred to as optimisers.

One disadvantage of optimisers using stochastic gradient descent alone is that they need many steps to converge as parameters get updated only in small steps, and each step is independent of the previous ones. This problem can be solved by introducing momentum, which can be interpreted very closely to the physical momentum when the loss landscape is interpreted as a physical potential. In this interpretation, the gradient descent introduces a force and the momentum mechanism adds a mass as well as friction to the system. The resulting behaviour is illustrated in Figure 3.

Fig. 3: Illustration of stochastic gradient descent, without momentum (left) and with momentum (right). Figure taken from [5].

Concretely, this means, we introduce a momentum ν at each step s, where

$$v^{(s)} = \alpha v^{(s-1)} - \eta \nabla_{\omega^{(s)}} L, \tag{14}$$

and a low value for the parameter α corresponds to large friction. Then, the parameter update becomes:

$$\omega^{(s+1)} = \omega^{(s)} + v^{(s)}. \tag{15}$$

Many modern optimisers implement different variants of momentum to reduce the number of steps needed for convergence. An overview can be found in the literature (e.g. in Refs. [1,5]) and visualisations are available online, e.g. in Ref. [6]. A key ingredient to gradient-based optimisation in all cases is access to the gradients. While in principle the gradients could be calculated numerically at each step, the amount of parameters in a typical deep neural network makes this inefficient and not feasible. Instead, gradients are calculated analytically in modern machine-learning frameworks, but typically without the explicit need of a user to do define their calculation manually. This is referred to as auto-differentiation and has contributed significantly to the success of deep learning approaches [7]. In auto-differentiation, each operation (e.g. the matrix multiplication in a dense neural network layer) is also assigned to an analytic gradient. Taking for example a very simple neural network

$$\Phi(\omega, x) = \tanh(\omega x) \tag{16}$$

and a loss function

$$L = (\Phi - y)^2, \tag{17}$$

Considering each operation separately, we also define $a = \omega x$. Then gradients for ω can be calculated based on the chain rule as:

$$\frac{\partial L}{\partial \omega} = \frac{\partial a}{\partial \omega} \frac{\partial \Phi}{\partial a} \frac{\partial L}{\partial \Phi} = [x] \cdot \left[1 - \tanh^2(a)\right] \cdot [2(\Phi - y)] \tag{18}$$

At a certain step of the optimisation, this analytic expression only needs to be evaluated for a certain numerical choice for x, y, and ω. This can easily be done by first performing a *forward pass* in which the neural network is evaluated, while saving the individual outputs for each operation: a, and Φ. Now the gradient with respect to ω can be calculated by evaluating the above expression, using those intermediate results. Moreover, it can be seen from the equation that even if x were an output of a previous neural-network layer, one could calculate the gradient for ω first when going backwards, and then use the numerical output of that to calculate the gradients for the weights in this hypothetical

previous layer. This process is called the *backpropagation*. It makes calculating gradients in large and complex neural networks computationally feasible and offers a simple way of calculating gradients by attaching analytical derivatives to fundamental operations only [7].

For a training to converge within a finite (best short) time, many parameters such as the learning rates, but also momentum parameters need to be tuned. With modern optimisers, the momentum parameters are often chosen well and require less tuning. The learning rate, however, depends strongly on the model complexity, the batch size, the loss function, and the training data. In practice, it needs to be tuned for each model by observing the loss value during the training. The training is typically performed in multiple iterations over the full training data set, referred to as an epoch.

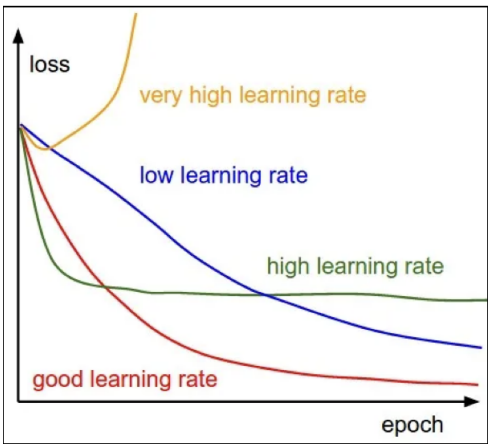

Fig. 4: Sketch of the behaviour of the loss during training for different learning rates. Figure taken from [8].

As illustrated in Figure 4, a too low learning rate can increase the training time significantly (and can also lead to the optimisation process getting stuck in local minima), while a too high learning rate may lead to missing the minimum or even no convergence at all. These effects can be intuitively understood with the help of Figure 5.

Since there is no generally applicable "best" learning rate, it is advisable to test a few learning rates in practice to gauge e.g. which learning rate is too high and at which rate the training stagnates. Typically, learning rates do not exceed 10^{-2} and are often in the range of 10^{-4}, depending on the neural network architecture. Also the best choices for parameters such as the number of nodes at each layer or the depth of the neural network need to be tuned by hand or by using other optimisation algorithms outside of the training procedure. These parameters are also called *hyper parameters*.

As mentioned before, deep neural networks can serve as universal function approximators. In case the neural network has sufficient expressivity (e.g. by having a large amount of free parameters), it can in principle learn each individual sample in the data set it is trained with. In general, this is not desirable, since it can (but curiously does not have to) imply that the network does not generalise, referred to as overtraining or overfitting. For the HEP context, that would mean it would learn to identify e.g.

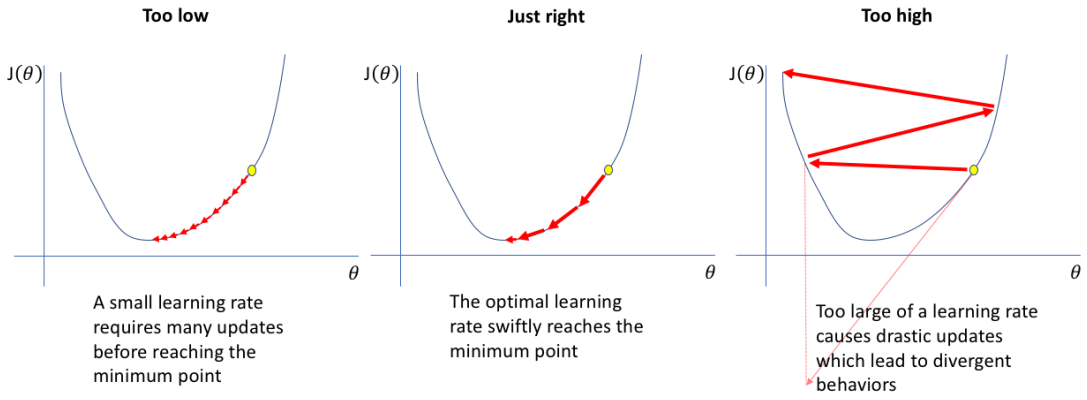

Fig. 5: Sketch of the behaviour of the loss during training for different learning rates. The loss value is represented by $J(\theta)$, and the parameter to be optimised by θ. Figure taken from [9].

individual collision events, but would not learn generic properties of the underlying physics process. If such a network is then confronted with new data, it would not give the desired result. To identify such behaviour, one usually defines a *validation* data set. In most cases, it is split off from the training data set and consists of samples that are never used in the training. By monitoring the loss in this validation data set during training, overtraining can be detected by an increase in the loss for the validation data set while the loss for the training data set still reduces. There are multiple ways to reduce overtraining, such as different regularisation techniques that also concisely described in Ref. [1] and others. Here, I would like to focus on the simplest approach: more data samples per neural network weight. This means either reducing the network complexity or creating more training data. While the latter is a large challenge in computer science, where many data sets rely on humans to assign a truth label (e.g. if a picture contains a cat or a dog), the situation in HEP is usually different. Here, we would usually train the models on simulated events that we can produce with very little effort in comparison. Therefore it is generally easier to produce a larger data set than studying different regularisation schemes involving a large set of additional parameters to be tuned. The same should be mentioned with respect to data augmentation and other techniques that address the lack of readily available training data. Often they matter less for HEP tasks than in computer science.

2 Exploiting the structure

The most effective way to control the parameter count while keeping expressivity is to exploit the structure of the data. That means if the data to be processed has certain known features or symmetries, the neural network should be adapted to that. The MLP that has been covered so far does not have this capability. It always learns the most generic approximate function based on a set of inputs. While it helps, for example, to sort the reconstructed particles in an event and their variables (or features) before they are fed to the MLP always in the same way, the MLP itself has no structure that in itself accounts for the fact that each particle is a representation of the same physical concept, but possibly with different features. As a universal function approximator, the MLP can learn this connection, however this may require more free parameters and more training (and in turn more computing resources). In some cases, the amount

of free parameters that would be needed becomes strictly prohibitive: for example on images. A typical camera e.g. in a smart phone has 10 to 50 megapixels. If each of these pixels would be taken as input to an MLP, and assuming 10 nodes in the first layer, the first weight matrix would contain 100 to 500 Million free parameters. In addition to the large parameter count, the compression to only 10 nodes in the first layer would not provide enough expressivity to perform even simple object identification tasks with such a network. Moreover, if the image was shifted by only one pixel, it would represent a completely unknown input to the neural network even though it shows the same object.

2.1 Convolutional neural networks

A solution to the problem induced by the shifts (and as discussed later for the large parameter count) is to implement filters that can slide over the image. Taking as example the task of identifying an image to show either a cat or a dog, such a cat filter could search for a cat in the image irrespective of the cat's position in the frame. More concretely, such a filter would compare the observed pixel configuration in each part of the image with patterns that could resemble the cat.

Convolutional neural networks (CNNs) [10, 11] implement such filters while keeping the amount of free parameters manageable by (a) learning filters from examples and (b) abstraction. Learning the pixel pattern to look for can be accomplished by a neural network. However, this neural network only considers a fraction of the pixels as inputs: those within the frame the filter is applied to, also referred to as the kernel size. Within such frame, the kernel itself resembles a dense neural network layer, where each pixel corresponds to one input node. This very same layer is applied every time the kernel is shifted. The output of this procedure produces a new image, where each pixel contains the output of the shifted kernel.

2.1.1 Learning filters

In the following we will construct a convolutional kernel step by step, first for a black and white image with one filter:

$$y_j = \theta\left(\sum_i^{N_k} \omega_i \, x_{I(j,i)} - b\right) \tag{19}$$

Here, the value of output pixel y_j is determined by the weights ω (for the moment each ω is strictly a scalar quantity) applied to the pixel values x of the neighbouring pixels. An index mapping function $I(j,i)$ determines the global index of a pixel given a global index of the output pixel j and a relative index within the kernel i. Such a kernel, acting on an input image without bias or activation, is also depicted in Figure 6.

Even though this is a very simple kernel, the main features of a CNN layer already become visible: ω are strictly relative, and each ω is applied and trained multiple times in the same image. This can be extended easily to multiple output nodes (in total N_F), also referred to as output channels:

$$y_{j\alpha} = \theta\left(\sum_i^{N_k} \omega_{i\alpha} \, x_{I(j,i)} + b_\alpha\right), \tag{20}$$

where α determines the output channel. The situation changes slightly when multiple input nodes (also

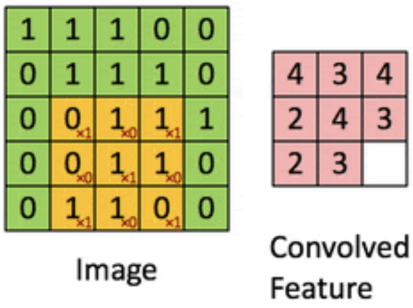

Fig. 6: Illustration of a single filter CNN kernel without bias or activation acting on an input image. The kernel is visualised as red numbers and its size is indicated by the yellow area.

referred to as input channels) are allowed. Then, additionally the input channel (β) has to be accounted for:

$$y_{j\alpha} = \theta\left(\sum_{\beta}^{N_C}\sum_{i}^{N_k}\omega_{i\alpha\beta}\, x_{I(j,i)\beta} + b_\alpha\right). \tag{21}$$

Here N_C is the number of input channels. This equation can be rewritten in terms of matrix multiplications (by – as an exception – introducing vector arrows here):

$$\vec{y}_j = \theta\left(\sum_{i}^{N_k}\omega_i\, \vec{x}_{I(j,i)} + \vec{b}\right), \tag{22}$$

where each ω_i is now a matrix of dimension $N_F \times N_C$ and $\vec{x}_{I(j,i)}$ a vector of dimension N_C. If we unroll the sum further, making ω a matrix with $N_F \times N_k \cdot N_C$ rows and $\vec{x}_{I(j)}$ a vector of $N_k \cdot N_C$ entries, determined by $I(j)$, we arrive back at an expression similar to the dense layer from Equation 1:

$$\vec{y}_j = \theta\left(\omega\, \vec{x}_{I(j)} + \vec{b}\right). \tag{23}$$

However, the relative nature of the convolutional kernel is still expressed by the pixel index j and the fact that also $\vec{x}_{I(j)}$ is built by selecting pixels relative to j through $I(j)$.

In this form, the connection to a discrete convolution, defined by:

$$(f * g)[n] = \sum_{m=-\infty}^{+\infty} f[m]g[n-m] \tag{24}$$

is not as clearly visible. However, one can relate the expression above to a convolution. The main part is to change the perspective of the term $\omega_i\, \vec{x}_{I(j,i)}$ in Equation 22 to refer to all N_p pixels rather than just the kernel:

$$y_j = \sum_{m=1}^{N_p}\omega(j,m)x_m, \tag{25}$$

where $\omega(j,m)$ returns the weight for global pixel index m within the kernel, if m is within the frame around j and zero otherwise. This can also be expressed to be a function of the difference between j and

m, and therefore it can be related to a convolution as follows[1]:

$$y_j = \sum_{m=1}^{N_p} x_m \, \omega(j-m). \tag{26}$$

As the CNN layer is indeed equivalent to a convolution, the property of *translation equivariance* also follows directly from it since translation and convolution operators commute, meaning that applying a pixel (or coordinate) shift and then the CNN layer returns the same output as applying the same CNN layer and then shifting the image. This should not be confused with translation invariance, where the output of the operation does not change with the translation. The latter property can apply, for example, to the whole neural network that classifies cat versus dog pictures (that will likely contain also CNN layers).

Special care needs to be taken at the edges of the images, that we so far neglected. As also indicated in Figure 6, the CNN layer would reduce the image size as the kernel cannot extend beyond the image boundaries. In some cases, however, the output should have the same size as the input (for example when denoising an image). In this case, one can apply a simple padding to the edges, where pixels with a fixed value (usually zero) are added such that the application of the CNN layer does not reduce the image size.

2.1.2 Abstraction and pooling

Even if a filter does not need to process the whole image, a kernel that is able to identify e.g. a cat would still need to cover a large area of the image, leading to a large kernel size, and that resulting in a large number of free parameters: $N_C \cdot N_F \cdot N_k$. In CNNs, this is kept under control through abstraction and pooling. The kernel sizes at each CNN layer are kept small, e.g. the first layer would only identify edges in an image, changes from one colour to the other, in very localised areas. Before the next layer, one would apply a pooling operation on this resulting image. A pooling operation summarises the information of neighbouring pixel groups by taking the mean or maximum value (the latter is more commonly used) for each feature over that pixel group. This way, the information contained in the image is compressed more and more. At the same time, this introduces abstraction into the CNN, since now the next layer would combine edges, represented by hidden vectors per pixel, into more complex objects, such as a cat's eye. Then subsequent layers could identify two cat eyes (accompanied by the corresponding mouth and nose) as a cat face etc.

Without introducing a large amount of free parameters in the early layers, this architecture gives rise to a large *receptive field* of the deeper layers of the network, illustrated in Figure 7, which finally allows a kernel in a deeper layer to "see" a whole object (in terms of the combination of all its parts) at once, and therefore identify it.

The architecture of a classification CNN is shown in Figure 8. Even though it is an early example, it shows all the typical features discussed above: it contains convolutions over the input image or hidden

[1] N.B.: there is a subtle flip in the matrix ω in this step that can be fully absorbed by the fact that the parameters of ω are free learnable parameters. Technically speaking the convolution operation in neural networks is actually a cross-correlation operation.

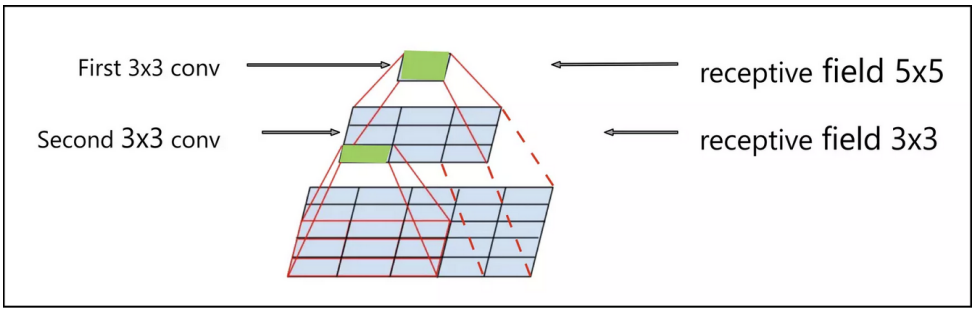

Fig. 7: The receptive field of a hidden layer in a CNN.

representations intersected with pooling operations, building a more abstract representation of the image. Since the final goal of the network is to identify hand-written numbers, the output of the last pooling operation is transformed into a simple vector (flattened) and passed to a dense neural network, finally classifying the number to be between 0 and 9.

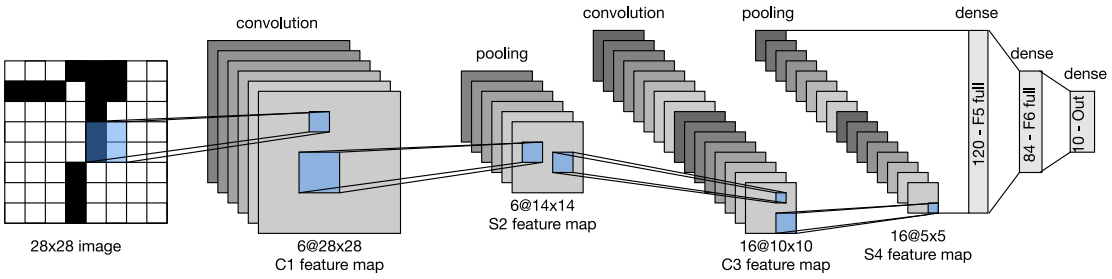

Fig. 8: "LeNet" [11].

To summarise, CNNs rely strongly on CNN layers. Each CNN layer is translation equivariant and therefore exploits the structure of image data. Moreover, the same weights are applied and trained multiple times within an image. This also means that the effective data set each weight can be trained with is up to multiple orders of magnitude larger than the number of samples contained in the data set, reducing the risk of overtraining significantly.

Examples of a direct application of a CNN in the HEP context can be found in Ref. [12] and many others. In particular the identification of the origin of jets is here a topic that has received a lot of attention at about the same time advanced DNN structures became used more extensively in HEP. In Reference [12], the energy deposits of the jet in the calorimeter are interpreted as an image. Since many current calorimeters can be segmented regularly in η and ϕ, a regular pixel grid can be built, centered around the jet direction, which is then fed to a classification CNN to identify the jet origin. Here, jets that stem from top quarks and jets that stem from lower-mass QCD interactions are being distinguished.

Figure 9 shows these jet images. In the preprocessed case, there is a clear distinction visible for the

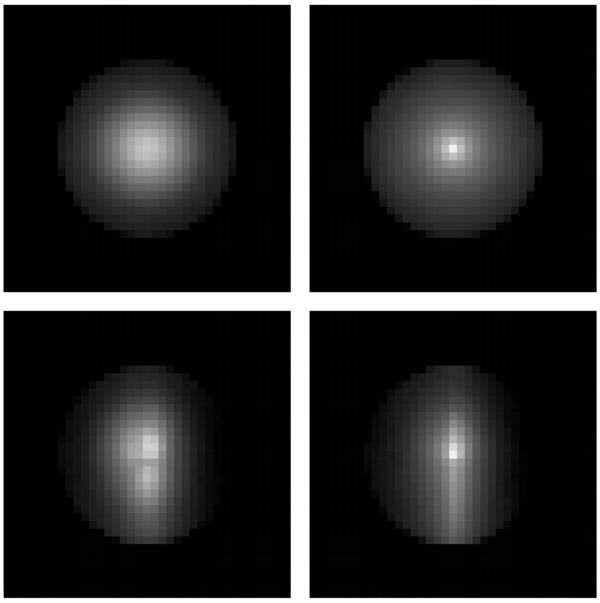

Fig. 9: The average of 100k jet images. The gray-scale intensity corresponds to the total transverse momentum in each pixel. Upper: no preprocessing besides centering. Lower: with processing (rotations and flips) ensuring the maximum intensity is in the upper right quadrant. Left: top-quark jets. Right: background jets. Figure taken from Ref. [12].

top-quark and the background jets that can be exploited by the CNN, improving the performance with respect to the case, where only high-level variables constructed by humans are used. A different way of using the structure of the data to identify jets - in this case stemming from b or c quarks - is incorporated into DeepJet [13]. This jet identification algorithm uses the full set of jet constituents, individually reconstructed first using all subdetectors. Each of these particle candidates is then processed by a per-particle neural network, which can be interpreted as a CNN layer with a kernel that operates on only one pixel (= particle) at the time, before the output is flattened and fed to a feed-forward DNN [2]. Exploiting the structure of the data in this way led to a tremendous improvement of the physics performance, in particular at high transverse momenta, compared to previous MLP approaches that needed to restrict themselves to selecting a few most significant jet constituents as they did not directly mirror the data structure.

2.2 Attention and transformers

In particular in HEP, most of our data does not come in the form of a regular grid - or is easily transformable to one. However, a regular grid, and some sort of translational symmetry on that grid, is needed to use CNNs. A prominent example from outside of HEP for data that does not follow a regular grid structure, but is also not fully unordered, is language. Sentences are sequences, where the order of the words determines their meaning, together with their individual meaning. So to match, for example, an image of a bear looking at a trout in a river to the sentence “The little bear saw the fine fat trout in the brook.”, a CNN for the image, and a mechanism to interpret the written sentence is needed. Also here,

[2] N.B: Also a recurrent neural network layer is used in-between the per-particle network and the feed-forward part, however the per-particle network was the main enabler

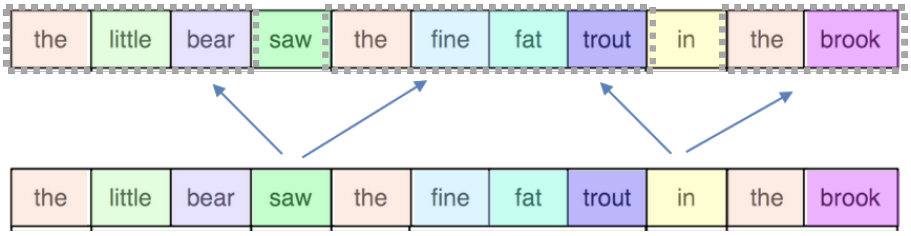

Fig. 10: Example relations between words in a sentence. The embeddings are symbolised by different colours, and the relations by arrows.

the structure of the data is the key. For sentences, usually the following applies:

- they are made of words, not pixel-values
- they do not have a fixed length
- the relations between the words matter
- there is a direction

In the following, we will go through these structural properties and how they are addressed in language-processing models starting from the CNNs to finally arrive at the *attention* mechanism, which is at the heart of the *transformer* architecture, the basis for powerful models such as chatGPT.[3]

In order to transform words into numbers that can be processed by a DNN, an *embedding* is performed, which is simply a translation of words (or parts of words) through a dictionary to vectors of real numbers. These embeddings can simply be learned by a shallow neural network that translates an index in the vocabulary (a number assigned to each word) to a vector in a higher dimensional space that is normalised to have a length of one to provide reasonably normalised input to the following part of the neural network. In addition to a word embedding, also a position embedding can be performed that would be added to the input feature vector of each word. More details on position embeddings can be found e.g. in Ref [14], but are not discussed in detail here since they are less relevant for the HEP context.

What is similar between an image and a sentence is that often words that relate to each other are actually close together in the sequence. For the issue of a fixed length one could - in principle - add zeros up to a maximum fixed length. This would allow to use the concepts of CNNs also for sentences: filters, abstraction, and summary building (pooling). However, while this may work for the short example here, it will not work for very long sentences, in particular for languages where words that relate to each other do not have to be close to each other in the sentence. However, it is possible to extend this idea in the sense that if it is not the nearest neighbours that are necessarily relevant, then a way to determine the relevance of the neighbours could be by the relation of one word to the other words in the sentence. This is illustrated in Figure 10.

So similar to a CNN layer, where the value of output pixel j is determined by the input values x of pixels in its neighbourhood (see e.g. Eq.22), one can define an operation here, that takes into account

[3] This is a very non-historic approach and often taught differently in the literature.

the relation a between the words to define an output value:

$$\hat{y}_j = \sum_i^N a(i,j) x_i. \tag{27}$$

This relation should depend on the words themselves, and possibly the position of the word in the sentence, in other words the input features of each word. Therefore, we can substitute $a(i,j)$ with $a(x_i, x_j)$. In addition, this *attention* should be directed as one for example the word "in" may need to relate to "trout" differently than "trout" would relate to "in". This can be accomplished by defining a function $k(x)$ that creates a *key* from the word features and a function $q(x)$ that creates a *query* from the word features. Making these functions learnable (e.g. by introducing a simple weight matrix), the attention can be expressed as $a(k(x_i), q(x_j))$. In addition, the attention should be scalar, while the function k and q are more expressive if they are vector-valued. This leads to the final definition of the attention (often referred to as dot-product attention):

$$a(i,j) = \underset{i}{\sigma}\left(k(x_i) \cdot q(x_j)/\sqrt{d_k}\right), \tag{28}$$

where $k(x_i) \cdot q(x_j)$ is the scalar product between the vectors $k(x_i)$ and $q(x_j)$, also referred to as *alignment*. The function $\underset{i}{\sigma}$ is the softmax function:

$$\underset{i}{\sigma}(\xi) = \frac{\exp(\xi_i)}{\sum_j \exp(\xi_j)}, \tag{29}$$

ensuring that the sum of the attention weights is one. The normalisation term $\sqrt{d}$ ensures that the variance of the scalar product remains roughly at one (with d being the dimensionality of the input vectors, leading to a variance of the scalar product of d for normal distributed vectors). The full expression then reads as:

$$a(i,j) = \underset{i}{\sigma}\left(\sum_\alpha^{d_k}\left(\sum_\beta^{N_C}\omega_{\alpha\beta}x_{i\beta}\right)\left(\sum_\gamma^{N_C}\tilde{\omega}_{\alpha\gamma}x_{j\gamma}\right)/\sqrt{d_k}\right), \tag{30}$$

where ω and $\tilde{\omega}$ are the weight matrices for k and q respectively and N_C is the number of input features per word (analog to the number of input channels for a CNN layer). To add more expressivity, also the features of each word x are transformed before they enter the attention-weighted aggregation, using another weight matrix, defining a full attention operation [15]:

$$y_{j\alpha} := \sum_i^N a(i,j) \sum_\beta^{N_C} \hat{\omega}_{\alpha\beta}x_{i\beta} = \sum_i^N a(i,j)\, v_{j\alpha}. \tag{31}$$

The last term is often referred to as the "value" transformation, making the attention aggregation work on *key* (k), *query* (q), and *value* (v) inputs. The whole mechanism here expressed as equations is also illustrated in Figure 11. Since key, query and value are all derived from the word features themselves, this operation is also referred to as self attention[4].

[4]This has also historical reasons.

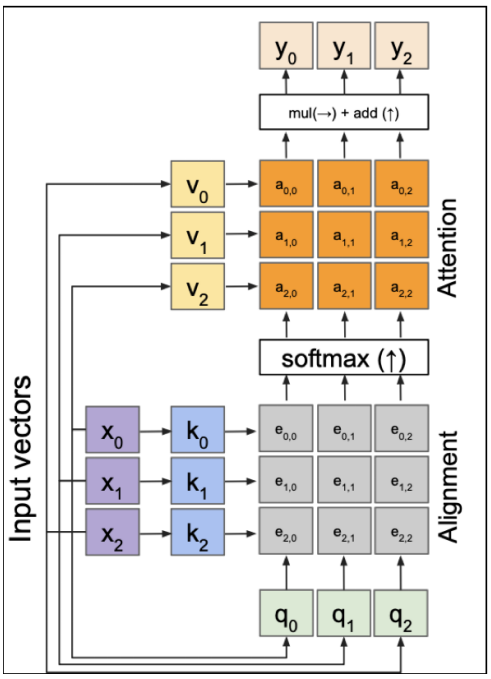

Fig. 11: Illustration of the self-attention mechanism.

Similar to defining multiple filters in a CNN layer, also here multiple *attention heads* can be defined, each attending to a different set of words, by repeating the above operation with independently learnable weight matrices. The respective outputs of the heads are then simply concatenated, meaning that the individual vectors, e.g. N vectors of dimension M, are then combined to a single vector with $N \cdot M$ entries. We can see how this mechanism could be immediately useful, e.g. when processing a jet with multiple constituents that do not follow a regular grid pattern, or other inputs in HEP. However, the most prominent application of the attention mechanism are transformer models.

A transformer is a neural network architecture that is built for sequence processing and revolves around the application of attention blocks. Due to its current prominence, there are many excellent sources on the internet that explain it very well, which have partially animated graphics, one of those being Ref. [16]. Therefore, the description here is kept a bit shorter and points out in particular those details that can also help improve the neural network performance in other architectures. A general overview of the architecture is given in Figure 12. The transformer model consists of an *encoder* block, that transforms the input to a hidden representation, and an *auto-regressive decoder* block that serves two purposes: it produces an output probability for the next word to be produced based on the encoded input text (e.g. when translating text from one language to another) and reads back its own output in an auto-regressive manner to produce the following words, conditioned on the fact that it has already produced the first (or a set of) words. It is important to note that in the connection between encoder and decoder, the keys and values of the encoder are fed to the second multi-head attention block of the decoder, such that the decoder can query different parts of the encoded input depending on the words that have already been generated.

Most building blocks of the transformer architecture in Figure 12 were already discussed. However, the "Add & Norm" blocks require additional explanation, in addition to the *skip connection* that

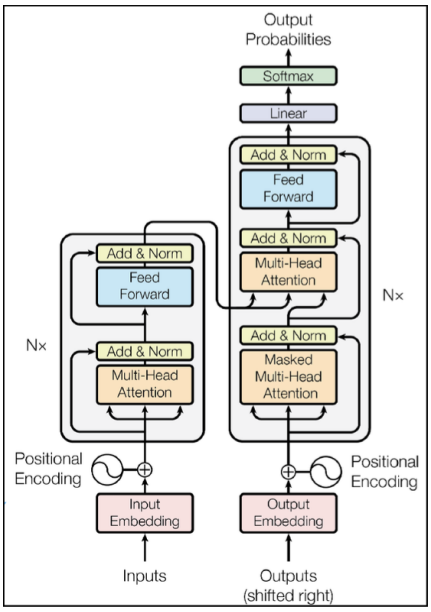

Fig. 12: Illustration of the original transformer architecture, taken from Ref. [15]. The "Nx" stands for the possibility to repeat the respective block multiple times.

connects the inputs and the outputs of each multi-head attention layer. A skip connection is a powerful tool when designing neural networks. It allows the information to flow through the operation that is skipped, but also to skip it. If, as in the case of the transformer, the output of the operation (here the multi-head attention block), is added to the original features, the operation only adds corrections to the original features, which is often easier to learn in practice. Moreover, it aids the training as there is always a direct connection of the gradients from the loss value to each of the learnable weight matrices in the backward pass. In consequence, it is much less likely that gradients vanish or explode. In addition, a *layer normalisation* is applied in the transformer model, which normalises each feature vector to have a magnitude of one, which can also help the neural network to learn faster and more reliably.

For sequence processing, the sequence is then fed to the transformer model, concluded with an *end token*. An end token is part of the vocabulary that marks the end of a sequence. Once the information is processed, the decoder part starts creating outputs. Here, the most probable word is chosen and passed back to the decoder into the auto-regressive input. Then the decodes produces the remaining part of the output sequence iteratively until an end token is produced by the decoder, at which point the generation ends.

A rather recent application of a transformer model in a HEP context is targeting jet identification, which remains a challenging task. The developments are heavily aided by readily available data sets to train and evaluate the models on, however I expect less and less relative improvement from newer models as the existing ones are already exploiting the information available to them quite well. The application to jets in HEP is called Particle Transformer [17]. The constituents of the jet are here interpreted as words in a sentence, each with its own features such as momentum and particle type. The final output of the architecture is a prediction of the particle that originated the jet, so a single (vector valued) output encoding the jet class instead of a sequence in the original transformer model. This is achieved by passing exactly one (empty) class token to the decoder part to create the queries of the decoder to the encoded

inputs finally leading to the prediction of the jet class. There is second main aspect that differs from the original transformer model: in the particle transformer the alignment matrix (the dot product of key and query pairs) is altered. In addition to fully learned key and query pairs, the physical relations of pairs of constituents in the jet, such as the angles between them, are added before the softmax function is applied. This decreases the mis-identification rate at a fixed efficiency to correctly identify the jet by up to a factor of almost two. Therefore, the particle transformer network is also an example that adding structural information (in particular physics-motivated information) to the neural network can help improve the performance - and with it the final physics reach - significantly.

2.3 Graph neural networks

In addition to image-like data and sequences, data in HEP can also come in other forms. One of them is the form of a graph. Formally, a graph consists of a set of points, also referred to as nodes or vertices, and connections between them, referred to as edges. These edges can either be undirected or directed. A prominent example for an undirected graph is the Facebook friend network, where friend requests have to be accepted by both parties. An example for a directed graph is the Twitter network, where person A can follow person B without B needing to follow A. This also illustrated that in a graph vertices as well as edges can have properties. Therefore a graph is an abstract way to describe "things" and relations between them.

In order to define learnable operations on such a graph a few things need to be considered. A graph does not provide a particular ordering of its content, therefore sequence processing operations cannot be applied directly. There is also no regular grid structure in a graph, so also CNN-like approaches cannot be applied, and finally a graph does not need to be of fixed size so even if it were computationally feasible, a simple MLP is also not the right tool to process graph information.

However, the graph itself provides a structure that can help define operations on it: the simplest operation is to independently update edge or node information, without any information exchange across the graph. In addition, information can be exchanged along edges between nodes, in a directed or undirected way, and possibly influenced by the edge properties. Moreover, edges or nodes can exchange information with a global entity. These paths of information exchange are summarised very well in Ref. [18]. Since the graph nodes and edges have no particular ordering, the concrete implementation of these operations should respect order invariance. This limits their set basically to a sum (or mean), a product, or minimum and maximum operations.

The simplest information exchange in a graph, where nodes are connected to other nodes is the following:

$$y_j = \underset{i \in N(j)}{\square} x_i, \tag{32}$$

where node j is updated by taking the sum (or mean), product, minimum, or maximum (here symbolised by $\square$) over the set of connected nodes $N(j)$. Another way of expressing this operation is using the *adjacency matrix* A:

$$y_j = \underset{i}{\square} A_{ji} x_i. \tag{33}$$

Here $A_{ij} = 1$ means that two nodes are connected and the $\square$ is applied over all nodes. More generally

this *message passing* can be expressed as:

$$y_j = \underset{i \in N(j)}{\square} \Phi(x_j, x_i), \tag{34}$$

where Φ is a generic function or a neural network. In this form, this reminds strongly of the attention mechanism introduced earlier, and indeed an attention layer can be interpreted as a weighted information exchange between nodes, with edge weights given by the attention between the nodes. Even a CNN can be phrased as a graph neural network, where the edge weights are learned and depend statically on the spatial relation of the central node (pixel) to the neighbouring nodes (pixels) - with the exception that here there is a specific ordering that can be exploiting directly. This illustrated that the graph formalism is very powerful and many conclusions drawn here apply directly to other neural network types.

In the HEP context, however, often there is not information to build a complete graph. In particular the edges are often missing. For example, the constituents in the jet or generally reconstructed final state particles in an event a-priori do not have connections between them. The same is true for hits in a detector, where a particle induced a signal above threshold, which have a position and other properties but no connections that would define a graph. This data mostly resembles a *point cloud*, a set of unordered points with properties.

So in these cases, a graph needs to be defined before operations can be applied on this point cloud. A simple solution would be to connect each point with all other points. This can only work in very simple cases, where the number of input points, N, is low, since it makes the number of connections grow by N^2, which makes this approach quickly resource-prohibited. Another approach could be to connect all points within a certain radius in the physical space. But also here, it is unclear if the connections that are defined are actually optimal for solving the problem. It can easily lead to too many connections or to few connections that can result on limited information exchange across the graph. It has been shown even formally that even a deeper graph with more iterations of information exchange cannot mitigate the issue of information-passing bottlenecks introduced by non-optimal connections [19] (and earlier references therein).

In the following these notes will focus on two examples of neural networks with learnable connections that optimise the graph topology during training. One of them is a the dynamic graph convolutional neural network (DGCNN) [20]. Input to the DGCNN architecture are a set of points making up the point cloud that may contain additional properties. Each of these vectors of point features is first transformed by a point-wise MLP, building a higher dimensional representation of each point. Based on this input there are two key features in the DGCNN: first, k nearest neighbours (in the original paper around 64) are selected using the euclidean distance between the points in full feature space; then the *EdgeConv* operation is applied:

$$y_j = \underset{i \in N(j)}{\square} \Phi(x_i, x_j - x_i), \tag{35}$$

where Φ is an MLP and for $\square$ the maximum is used. The key element here is that the operation processes the relations between the points, in other words the edge properties, where the edge properties are $e(i, j) = x_i - x_j$. Depending on the task, either the node properties are then aggregated into one global descriptor (a global vector built by maximum pooling over all nodes) and this global vector is used as

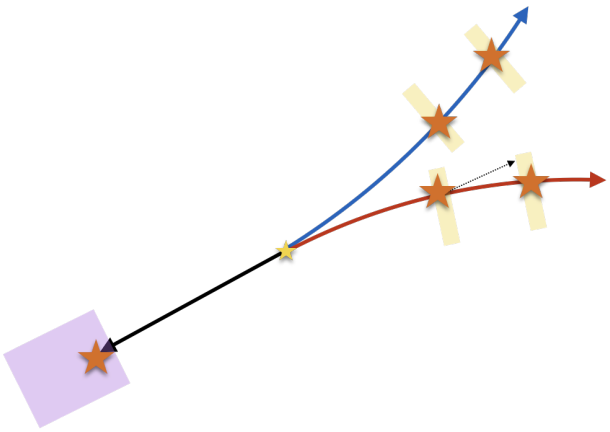

Fig. 13: Sketch of particles produced in a central collision passing different detector elements. Different particles are indicated by red, blue, and black arrows. A calorimeter is illustrated by the purple box, and tracking detector layers by yellow boxes. The interaction with the detector elements is symbolised by orange stars and the primary event with a yellow star.

input to a feed-forward DNN to perform a classification task of the entire point cloud, or the global vector is added to each node feature again, e.g. to perform a *semantic segmentation* task, where each point is assigned to be part of a certain class of object (e.g. of a car or a pedestrian).

This architecture is very powerful and has been adopted for many applications in HEP, such as ParticleNet [21]. ParticleNet is equivalent to the DGCNN architecture modulo small modifications regarding the choice of a few hyper parameters applied to the identification of jets[5]. In comparison to, for example, the image-based jet identification algorithms, using the DGCNN architecture improved the performance drastically, owing to the fact that it captures the structure of the jet data, containing individual constituents of the jet rather than only images, better. Moreover, the EdgeConv operation, targeting the relation between the constituents in the jet directly, contributes to the improved performance, and is another example that exploiting the structure of the data almost always comes with benefits.

3 Examples for advanced applications in HEP

While many of the techniques discussed here target classification, e.g. the separation of a signal from a background process with feed-forward neural networks or the classification of jets, there are also other applications of machine learning in HEP. One of them is closely related to the classification task: the regression task, e.g. to regress a correction to the measured jet momentum. These are conceptually straight forward extensions of classification neural networks and the same considerations apply. However, there is another class of applications that are conceptually very different: the reconstruction of multiple objects from detector signals, also illustrated in Figure 13. On the one hand, these tasks require only a single output, or a fixed size output; and on the other hand, they pose additional constraints on the resources that are not as severe in other cases. Both will be discussed in the following.

A typical modern multi-purpose HEP detector such as CMS or ATLAS consists of multiple subde-

[5] the focus on jet identification in these lectures has two reasons: one is that a lot of advanced machine learning techniques have found their way into this area, and the other one is to keep consistency between the applications that allows to compare them given the limited scope of these lectures.

tectors, each equipped with a large number of possible read-out channels fed by active detector elements that can detect particles traversing them or being stopped. In total, such a detector can easily reach $(O)(10^8)$ sensors. With an occupancy in the range of per-cent for a typical event, this means $(O)(10^6)$ of such active sensors, also referred to as hits, need to be processed.

While the detector data lends itself to being interpreted as a point cloud, and graph neural networks are very powerful in extracting information from that point cloud, the large amount of input points needs to be considered in the choice of the neural network architecture and can easily pose prohibitive constraints on it. For example if a self-attention-based layer such as in a transformer would be chosen, N^2 attention weights would need to be calculated. With $(O)(10^6)$ points this is not feasible. Here, more local algorithms provide two advantages: they only consider a smaller amount of possible connections and are therefore more likely to be computationally feasible; and they also have a better chance of being robust when confronted with unknown (physics) data, since they are strictly local. Locality is an important feature in such algorithms since the reconstruction of one particle should not be affected by the reconstruction of another particle at the other end of the detector[6]. For example DGCNN could provide such a locality constraint as it only processes edges of nearest neighbours. As a reminder, the main operations in DGCNN are building K nearest neighbours in the feature space of dimension F, and processing the edge information between the neighbours, also of dimension F. That means, with N points as input, one iteration of EdgeConv subjects an MLP to $N \times K \times F$ inputs, where the MLP has multiple layers with roughly F^2 free parameter. For typical values that would amount to more than $100\,000 \times 64 \times 64^2$ parameters to evaluate. This poses an issue that is overcome by GravNet [22]. As a first step, a GravNet layer learns coordinates of small dimensionality S from the input features, and additionally a feature vector to facilitate the information exchange. The first advantage in terms of resources is, that only the low-dimensional coordinates are used to calculate the k nearest neighbours, bringing a factor of 10 improvement in terms of resource usage over DGCNN. The second advantage is that instead of processing the edge information, in GravNet features are aggregated by a weighted sum - without MLP operations on the edges:

$$y_j = \underset{i \,\epsilon\, N(j)}{\square} \exp(-d_S(i,j)^2) D(x_j), \tag{36}$$

where $d_S(i,j)$ is the euclidean distance between node i and j in coordinate space S. This implements a geometric attention mechanism with D being a dense layer creating the value to be passed to the attention operation. The Gaussian weighting ensures that small distances correspond to large attention. In this way, points that should exchange a lot of information are pulled together in the coordinate space. In turn this means selecting the k nearest neighbours in that space also builds an optimal graph for the task at hand, since finally the loss function will determine what the best path of information exchange is. Moreover, this phrasing avoids processing edge information by any learnable operation directly, leading to a reduction of resource needs with respect to EdgeConv of a factor $k \approx 64$[7]. The full sequence of operations is also illustrated in Figure 14.

The physics performance of this information exchange is comparable or at times even better than

[6] N.B: Locality is also important for calibration of said algorithms, their bias they receive from the training sample, and their interpretability.

[7] In practice this can become even larger due to the way algorithms are implemented

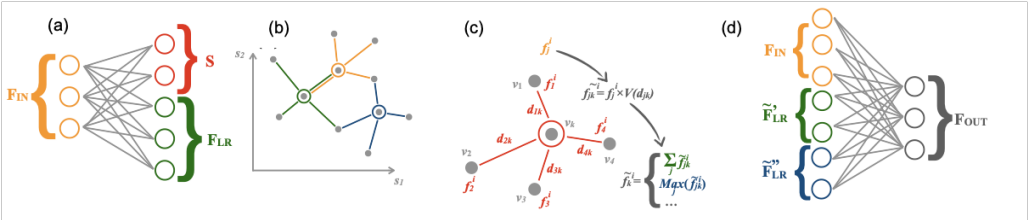

Fig. 14: Illustration of the steps in the GravNet layer. a) The input features are projected to coordinate features, S, and features to be exchanged between the points (F_{LR}. b) Nearest neighbours are connected in the space spanned by the coordinates S. c) The neighbour features are aggregated weighted by the distance in space S, taking the mean and the maximum of the weighted neighbour features. d) The aggregated information and the original inputs are fed to a dense neural network layer to build the final output of the GravNet layer.

the performance of DGCNN, for example when assigning hits in a highly granular calorimeter to one of two particle showers that stem from charged pions, or when assigning hits to different classes of objects in events, as in a study by the ATLAS collaboration [23]. This is noteworthy because it enables full event reconstruction with machine learning from detector hits in terms of computing resources, as well as that it showcases the strength of attention-based architectures together with the possibilities arising from learning the graph topology that facilitates the information exchange directly.

As just discussed network architectures exist for the reconstruction of multiple particles from detector hits, however these networks need to be trained, and individual objects need to be separated. This task is usually referred to as *instance segmentation* in computer science (not to be confused with *semantic segmentation*, where the aim is to determine the class of the object a certain point belongs to).

In HEP most computer vision techniques to perform such an instance segmentation do not apply directly [24]. This can be achieved, however, with object condensation [24], a technique that trains the neural network to convert a complex problem like separating the hits of different particles in the detector, that can overlap partially and have complex shapes, to a simple problem where the hits are reorganised in a learnable space and build clear ideally round clusters. In addition, also properties can be learned for each object such as the particle type or its momentum. Without going into too many technical details, the central point is to define a confidence measure β for each point and train it such that at least one point per object has a large β value and all noise points have low β values. Then, potentials are defined that create an attractive force for points belonging to the same object in the learnable *clustering space*, and a repulsive force between points that belong to different objects. To pair-wise loss calculation that would lead to $\mathcal{O}(N^2)$ relations to be calculated, only the highest β points per object are considered to act on all other points, but the remaining points do not directly interact with each other. These attractive and repulsive potentials created by each object are scaled with the β value of these highest points of the corresponding objects. They also scale with the β value of point that is either attracted of repelled, such that the gradient that is calculated to train the neural network creates a force scaling with β. In consequence, the highest β points will be in the center of the clusters in clustering space. Moreover, the network can also be trained to predict the object properties. They are predicted for each point belonging to the that object, and the corresponding loss is also scaled with β. As a result, the central highest β points per cluster also carry the best property estimate for the object. They are referred to as *condensation*

points. In a last step, the objects can be collected in clustering space by starting with the highest β point, assigning points around it (within a distance threshold) to that object and removing them from further processing, and moving on to the next highest β point until another threshold is reached. This simple last clustering step then ensures no object duplicates. This final step is illustrated in Figure 15. In particular visible is that in the clustering space, the cars are well separated from the background pixels and from each other. Even though the algorithms are not made for the task shown here to identify and segment individual cars in the images, they perform reasonably well, and are comparable to state-of-the-art computer vision approaches.

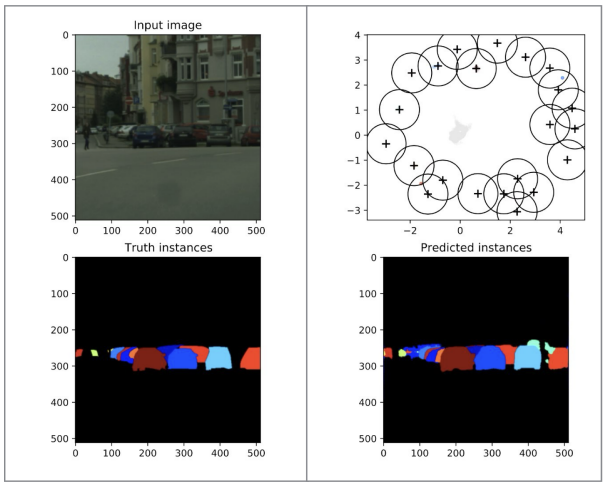

Fig. 15: Application of object condensation to a computer vision problem, identifying cars in an image. Top left: the original image, top right: the clustering space including the circles assigning points to each of the condensation points. Here, the background pixels are coloured grey. Bottom left: the truth instances, assigning each pixel to a particular car or the background. Bottom right: cars segmented by object condensation.

In the HEP context, this technique has also proven to be very powerful in context of calorimeter reconstruction (e.g. Ref. [25] and others), but also for track reconstruction [26]. Moreover, paired with neural networks like GravNet it allows creating a rather detector agnostic reconstruction that could be applied (after retraining) to very different types of detectors or problems.

Besides architectures that exploit the structure of the problem, there is a plethora of applications of such architectures for many purposes beyond classification or object detection. Of particular interest for physics are anomaly detection, often relying on auto-encoders, or even generative networks that can be used to generate images, but also as a fast simulation of the interaction of particles with matter, targeting future computing challenges. The underlying principles from a machine-learning perspective are described in a concise form in Ref. [1] and the references therein. Their applications to high energy physics are listed in the living review of ML for HEP [27].

4 Summary

Machine learning has left a tremendous impact and has enhanced the physics reach in high energy physics in the last years significantly. In most cases, this success relies on the exploitation of the structure of the problem (the physics) while giving sufficient freedom to the algorithms to learn also subtleties of the

data that are inaccessible for classic algorithms. This lecture aimed particularly at providing a tool set to mirror the physics structure in the network architecture, covering MLPs, up to attention based algorithms, and also provided a glimpse of advanced object detection in physics detectors.

References

[1] F. Fleuret, The little book of deep learning, https://fleuret.org/francois/lbdl.html.

[2] S. Dubey *et al.*, Activation functions in deep learning: A comprehensive survey and benchmark, doi:10.1016/j.neucom.2022.06.111.

[3] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.

[4] K. He, *et al.*, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV).

[5] I. Goodfellow, *et al.*, Deep Learning, MIT Press (2016).

[6] L. Ljang, A Visual Explanation of Gradient Descent Methods, https://towardsdatascience.com (2020).

[7] A. G. Baydin, *et al.*, Automatic differentiation in machine learning: a survey, arXiv:1502.05767 (2015).

[8] Stanford CS231n: Deep Learning for Computer Vision (MIT Licence), https://cs231n.github.io/neural-networks-3.

[9] J. Jordan, Setting the learning rate of your neural network (2018), https://www.jeremyjordan.me/nn-learning-rate/.

[10] K. Fukushima, Neural network model for a mechanism of pattern recognition unaffected by shift in position - Neocognitron. (1979), Trans. IECE

[11] Y. LeCun, *et al.*, Gradient-based learning applied to document recognition (1998), Proceedings of the IEEE 86, doi: 10.1109/5.726791

[12] S. Macaluso, D. Shih, D. Pulling out all the tops with computer vision and deep learning, JHEP (2018) doi: 10.1007/JHEP10(2018)121

[13] E. Bols, *et al.*, Jet flavour classification using DeepJet, JINST 15 (2020) doi: 10.1088/1748-0221/15/12/P12012

[14] K. Erdem, Understanding Positional Encoding in Transformers (2021), towardsdatascience.com

[15] A. Vaswani, *et al.*, Attention Is All You Need, NeurIPS proceedings 2017

[16] Jay Alammar, The Illustrated Transformer, jalammar.github.io

[17] H. Qu, *et al.*, Particle Transformer for Jet Tagging, Proceedings of ICML2022

[18] P. Battaglia, *et al.*, Relational inductive biases, deep learning, and graph networks, arXiv:1806.01261

[19] F. DiGiovanni, *et al.*, On Over-Squashing in Message Passing Neural Networks: The Impact of Width, Depth, and Topology, Proceedings of ICML2023

[20] Y. Wang, *et al.*, Dynamic Graph CNN for Learning on Point Clouds, ACM Transactions on Graphics (2019) arXiv:1801.07829

[21] H. Qu, L. Gouskos, ParticleNet: Jet Tagging via Particle Clouds, Phys. Rev. D 101 (2020) doi: 10.1103/PhysRevD.101.056019

[22] S. Qasim, *et al.*, Learning representations of irregular particle-detector geometry with distance-weighted graph networks, EPJC (2019) doi: 10.1140/epjc/s10052-019-7113-9

[23] ATLAS Collaboration, Physics Object Localization with Point Cloud Segmentation Networks (2021), ATL-PHYS-PUB-2021-002

[24] J. Kieseler, Object condensation: one-stage grid-free multi-object reconstruction in physics detectors, graph, and image data, EPJC (2020) doi: 10.1140/epjc/s10052-020-08461-2

[25] S. Qasim, End-to-end multi-particle reconstruction in high occupancy imaging calorimeters with graph neural networks EPJC (2022) doi: 10.1140/epjc/s10052-022-10665-7

[26] K. Lieret, *et al.*, High Pileup Particle Tracking with Object Condensation (2023), doi: arXiv:2312.03823

[27] HEP ML Community, A Living Review of Machine Learning for Particle Physics https://github.com/iml-wg/HEPML-LivingReview