JOINT INSTITUTE FOR
NUCLEAR RESEARCH

DZHELEPOV LABORATORY OF NUCLEAR PROBLEMS

DUBNA, RUSSIA

Final Report for the Summer Student Program

PRIMARY VERTEX RECONSTRUCTION FOR THE JUNO EXPERIMENT USING NEURAL NETWORKS

Submitted By:
Menno Door,
University of Heidelberg
and
Dmitry Aleksandrovich Selivanov,
St. Petersburg State University

Submitted To:
Maxim Olegovich Gonchar
and
Konstantin Andreevich Treskov,
Joint Institute for Nuclear
Research

June 25 - August 24, 2018

Abstract

This paper is adressing the primary vertex reconstuction in the JUNO central detector using neural networks. The main drawback of the charge center method is underestimation of the calculated radius. On the other hand, maximum likelihood method needs huge computational effort. Neural networks could be the answer to this problem. Two different approaches with neural networks are described in this work. First is the feedforward network which does correction of the charge center estimation based on summary data. Second is the convolutional neural network which use charge and time data of all photomultipliers by projecting the spherical structure of the JUNO central detector to the 2D plane.

This report is a collaborative work of two authors. The authorship of sections are marked with the initials of the author.

Contents

1 Introduction [MD]

The Standard Model (SM) is the currently most successful theory describing the fundamental particles that make up our universe and their interaction forces (except the gravitational force). Nevertheless the standard model is known to be incomplete. Questions of fundamental physics are still unanswered and several experimental findings imply physics beyond the standard model, such as:

- Asymmetry in baryon / anti-baryon abundance in the observable universe.
- Experimental hints on unidentified *dark matter* and *dark energy*.
- Experimental prove that neutrinos are massive particles.
- Lack of a unified theory of general relativity theory and quantum world.

The weak interaction, the only interaction neutrinos contribute to, is not well tested due to the fact that the probability of interaction is low (*weak*). In order to test the SM to new limits and identify new theories and symmetries it is necessary to study the currently unknown properties of neutrinos like their masses or at least their mass hierarchy, the equality of neutrino and anti-neutrino (Majorana-particles) and look for existence of additional light or heavy neutrino generations, i.e. sterile neutrinos.

Studies of neutrino physics have historically produced a large number of puzzles and mysteries such as deficit of solar neutrino flux, LSND, gallium and reactor anomalies. Some of them such as studies of deficit of solar neutrino flux led to discovery of neutrino mixing. Other anomalies can also indicate a gap in the Standard Model and lead to experimental discovery of physics beyond the SM.

Among the fundamental parameters that can be tested with neutrino oscillation experiments only phase of CP-violation and mass ordering or hierarchy remain unknown for now.

The accurate knowledge of neutrino mixing and oscillation parameters is important for testing the ground of SM. Future experiments such as JUNO, DUNE and Hyper-Kamiokande will provide measurements of CP-violating phase and mass ordering with unprecedented precision leading to the long awaited era of precision measurements in neutrino physics.

JUNO One of the future experiments addressing neutrino related studies is the *Jiangmen Underground Neutrino Observatory* (JUNO). JUNO is designed to study neutrino mass ordering and to provide measurements of three of neutrino oscillation parameters with sub-percent accuracy utilizing antineutrino flux from Yangjiang and Taishan nuclear power plants.

The detector will be located in Jinji town, Kaiping city, Jiangmen city, Guangdong province, 700 m underground at optimal distance of 53 km to nuclear power plants in order to achieve optimal sensitivity to neutrino mass ordering.

The JUNO central detector is a spherical multipurpose detector with 20-kilotons of liquid scintillator with a radius of about 19.5 m which is sensitive to electron antineutrinos

via the inverse beta decay (IBD) reaction $\bar{\nu}_e + p \rightarrow e^+ + n$. The surface of the acrylic sphere is covered with 18 000 large photomultiplier tubes (PMTs) of about 51 cm diameter. PMTs are placed in an hexagonal grid with occasional gaps for electronic ports and the chimney on a top for calibration source deployments. In small gaps between these large PMTs additional small PMTs with an diameter of about 3 inch are placed to increase the geometrical coverage and aid with muon reconstruction by providing more accurate timing information.

Such a detector would allow JUNO to perform multiple studies and measurements:

- Neutrino mass ordering determination with a significance of $3 - 4\,\sigma$;
- Measurements of mixing angle θ_{12} and mass splittings Δm^2_{21} and Δm^2_{32} with sub-percent accuracy;
- Collect unprecedented sample of geoneutrinos;
- Measurements of diffuse supernova neutrino flux;
- Searches for proton decay;
- Direct unitarity tests;
- Detection of solar and atmospheric neutrinos;
- Searches for non-standard neutrino interactions.

The most stringent requirement for experiment that should be achieved is energy resolution of $3\%/\sqrt{E\ (\mathrm{MeV})}$ or better. The sensitivity to correct determination of mass ordering drastically depends on it. In order to meet this goal sophisticated algorithms for vertex and energy reconstruction have to be developed. Additionally due to the amount of data which is going to be produced, these algorithms should be as fast as possible.

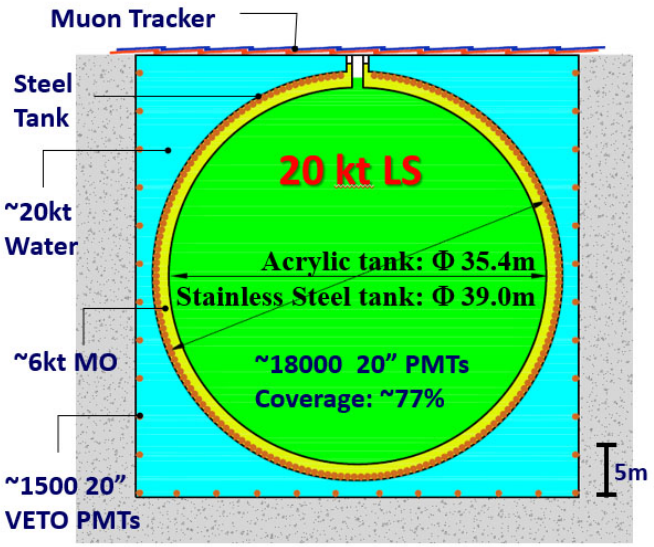


Figure 1.1: Simplified scheme of the JUNO detector.

2 Theory

This chapter will address the theoretical basics concerning the classical methods of vertex reconstruction for comparison and an introduction to neural networks.

2.1 Classical vertex reconstruction [MD]

Reconstructing vertices's of events inside a particle detector is classically done by plain mathematical methods and fitting algorithms. In the following, exemplary methods are shortly introduced for efficiency and accuracy comparison with the neural network methods.

2.1.1 Charge center method

The most naive and simple way to estimate the vertex of an event is the charge center method. The center of charge of an event inside the detector is given by:

$$\bar{\boldsymbol{x}} = \frac{\sum_i \boldsymbol{x} \cdot \boldsymbol{n}^i_{\boldsymbol{p.e.}}}{\sum_i \boldsymbol{n}^i_{\boldsymbol{p.e.}}} \quad , \tag{1}$$

with x being the position of the i-th PMT and and $n^i_{p.e.}$ the number of photo electrons emitted by the i-th PMT during the given event.

As an example, a set of about 156000 simulated Positron-events with an energy of 5 MeV and random vertex positions (using Monte-Carlo method) is analyzed using this method. The results shown in fig. 2.1 show a rising error in estimation with rising vertex radius. Especially at the far edge of the detector ($r > 16\,\mathrm{m}$) the calculated position is largely underestimated.

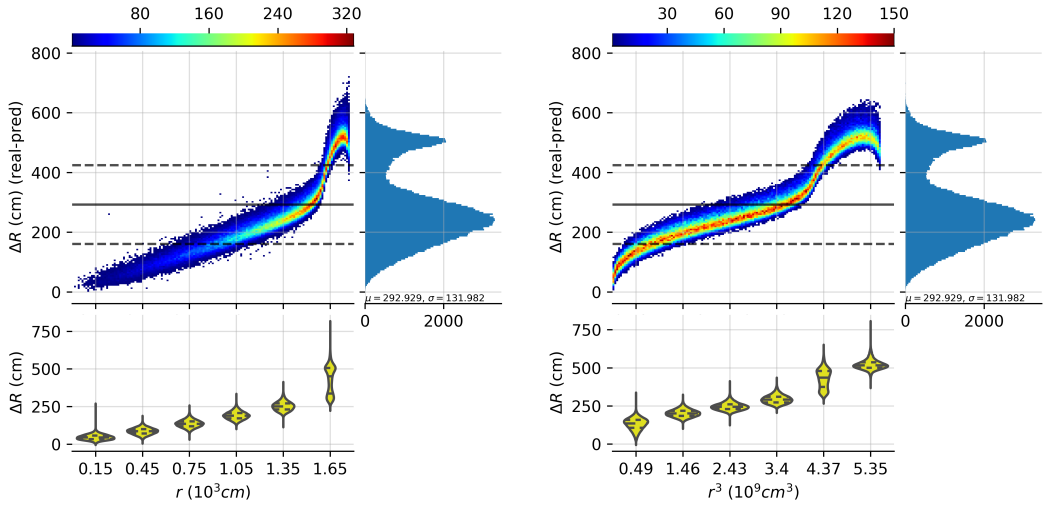


Figure 2.1: Example of vertex reconstruction using charge center method on 156000 5 MeV-events of Positron. Shown is the absolute distance between reconstructed and real vertex position over radius and cubic radius.

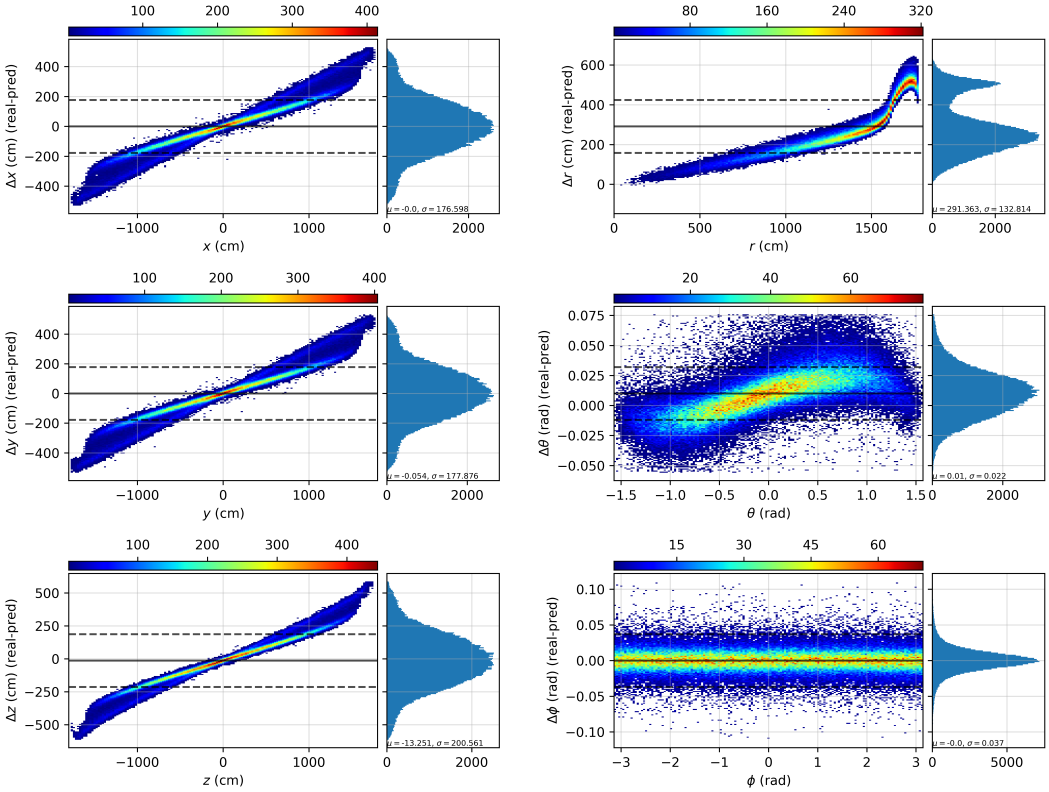


Figure 2.2: Example of vertex reconstruction using charge center method on 156000 5 MeV-events. Shown are deviations in cartesian and spherical coordinates of the reconstructed vertex position from the real vertex position.

An overview of the discrepancy in specific axis is shown in 2.2. The error of the radius estimation is the substantial contribution in the error of total distance. The error of the estimated longitudinal angle ϕ is stable over all radii and the standard deviation is $\approx$ 0.04 rad. The error in azimuthal angle θ is slightly varying a long the radius, probably due to the asymmetry of PMTs in this axis, but is still very accurate with a standard deviation of $\approx$ 0.023 rad. The error in the radial axis is quite severe with rising deviation up to 6 m at the edge of the detector. The knee feature at high radii is due to the flat angle of emitted γ vectors relative to the surface of near PMTs resulting in total reflection and final detection in more distant PMTs. This results in a huge underestimation of the calculated radius. The reason for the global slope is described in [1] and can be easily corrected with a factor of $\approx \frac{6}{5}$.

The systematic features in the results of this method may enable further processing for better estimation. In the scope of this program we will use the charge center results as input for a simple neural network.

2.1.2 Maximum likelihood method

The currently most accurate algorithms for vertex reconstruction in the central detector of JUNO are based on the maximum likelihood method. This method is based on model

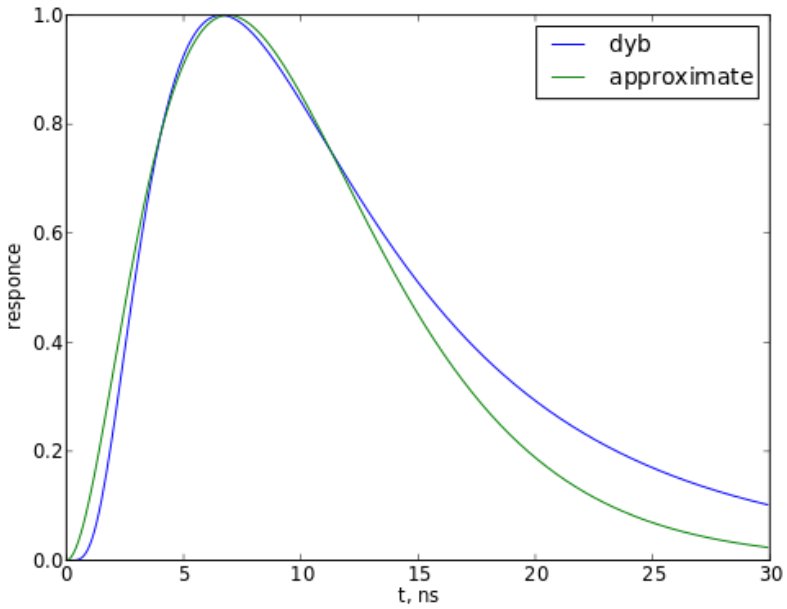


Figure 2.3: Single photon PMT signal at the Daya Bay detector.

functions which describe all physical interactions inside the detector and the behavior of the detection electronics. The vertex position is a parameter in the model function and the output of the function is the expected signal of the detector for this kind of event at the given position. In order to determine the vertex position of a given detector signal, the input parameters of the model function are varied until the output of reassambles the real detecotr signal in the most similar way (most-likely).

For JUNO the time information of PMTs combined with the optical model in the detector are used for these reconstruction algorithms which are able to determine the vertex position with a bias within 3 cm in fiducial volume and vertex resolution is 7 cm at 1 MeV [1].

The main drawback of these methods is the huge computational effort needed. A reconstruction of a single vertex may take several minutes. Neural networks could be the answer to this problem since they need an order of magnitude less time produce results after they have been sufficiently trained.

2.2 Neural networks [DS]

Machine learning is a set of algorithms used to solve severe and complicated tasks. Neural networks are a huge part of machine learning. The main goal of neural networks is to approximate some function $y = f(x, \theta)$ which maps an input $\boldsymbol{x}$ to a label $\boldsymbol{y}$. They tune parameters θ during training by minimizing a loss function value so that the network produces a desired output for a given input. Neural networks are called *networks* because they consist of layers with neurons inside each layer. We can think about neural networks as a chain of functions which are composed together:

$$f(\cdot) = f_3(f_2(f_1(\cdot))) \tag{2}$$

In this case f_1 is called the first layer, f_2 is called the second layer and so on. The essential terms of neural networks can be explored on an example of a *feedforward* network (fig.2.4).

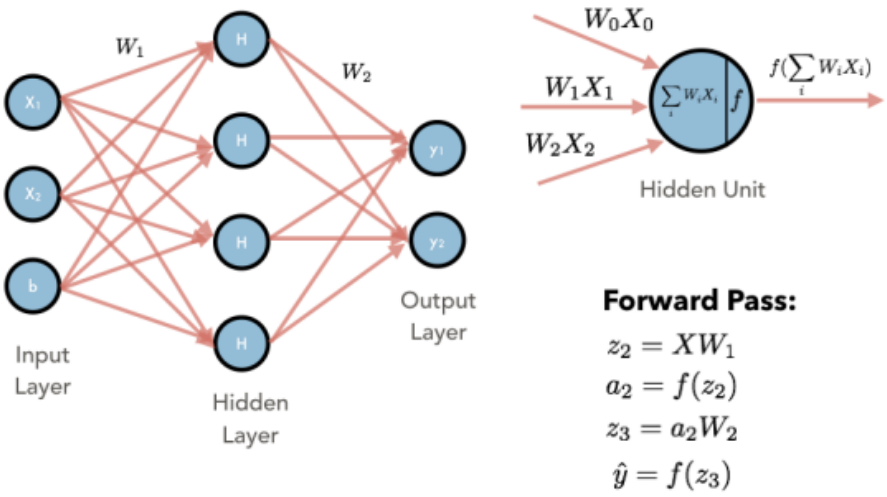


Figure 2.4: Simple neural network architecture example. This is a *feedforward network* (also called a *multilayer perceptron* model). All layers between the input and output layers are called *hidden*, because the training data doesn't show desired output for each of these layers. Each neuron in such layers is called a *hidden unit*.

The *output* of the i-th neuron in the k-th layer can be calculated by:

$$y_i^{(k)} = f_i^{(k)} \left(\sum_j w_{ij}^{(k)} \cdot x_j + b_i^{(k)} \right) \tag{3}$$

where $f_i^{(k)}$ is an *activation function* of this neuron, $w_{ij}^{(k)}$ is a *weight coefficient* of this neuron with respect to the j-th input, x_j is a value of j-th input (if k corresponds to the next layer after the input layer) or a value of j-th output of $(k-1)$-th layer (if k corresponds to other layers) and $b_i^{(k)}$ is a *bias coefficient* of this neuron.

In this way a set of variables to train θ consists of weights w_i and biases b_i. The most common used algorithm for minimizing the loss function (finding best set of parameters θ) is *gradient descent*, but there are a lot of these algorithms and what to choose depends on a concrete task. Deep networks have huge number of parameters to tune so we need to choose a method to compute gradients efficiently. *Backpropagation algorithm* serves this purpose [2].

In order to define a neural network we need to choose:

- Architecture (a type/number of layers, a number of neurons in each layer)
- Optimization method
- Loss function
- Activation functions for every neuron

- Hyperparameters

The loss function maps a difference between predicted and desired values onto a real number. We can choose loss function so that it intuitively represents some physical meaning, but it is a common practice to use such functions that represent some "cost" and have less operations to compute. In example, for regression tasks it is often the *mean squred error* function:

$$MSE = \frac{1}{n}\sum_i \left(y_i^{true} - y_i^{pred}\right)^2 \tag{4}$$

where y^{true} is a desired output vector of n components and y^{pred} is a predicted output vector of n components.

The activation function of a neuron defines an output of this neuron according to a given input or set of inputs. Only non-linear functions allow neural networks to solve complicated problems. Usually it is defined globally same for all neurons in a layer. Very popular activation function is the *Fermi function* or the *sigmoid function* (fig. 2.5):

$$f(y) = \frac{1}{1+e^{-y}} \tag{5}$$

which maps a range of values $(-\infty; +\infty)$ to $(0; 1)$. Also a commonly used function is the *hyperbolic tangent* function (fig. 2.5) which maps a range of values $(-\infty; +\infty)$ to $(-1; 1)$.

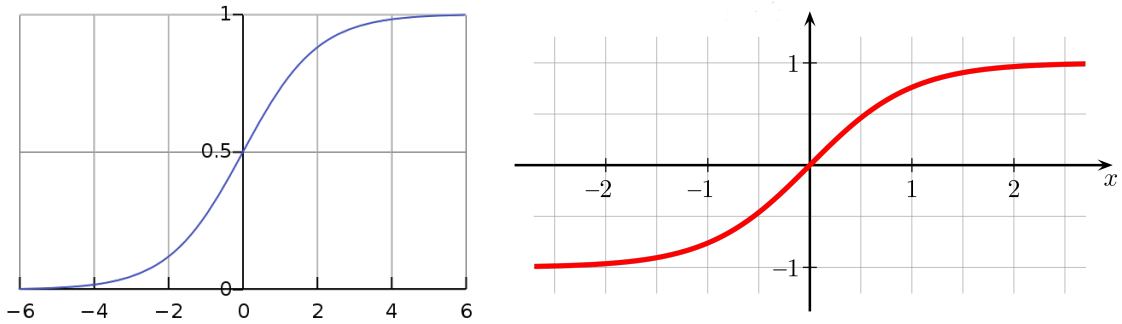


Figure 2.5: Sigmoid (left) and hyperbolic tangent (right) functions.

Hyperparameters are variables which determine how the network is trained. Examples include a learning rate for the optimization method, regularization scale factors (see 2.2.2 for details), a batch size, a number of training steps, a number of epochs, coefficients for local response normalization (see 2.2.2 for details) and etc. Also the number of neurons and layers could also be thought as a hyperparameters when we want to find the best architecture for solving a concrete task.

Unfortunately, if the dimensionality of an input data is large – it causes the huge number of parameters to train. In example, if we have a charge response of 20000 PMT's, 2 hidden fully connected layers with 20000 neurons inside each layer, 1 output layer with 3 neurons, then the number of only weight coefficients in a feedforward network would be:

$$20\,000^2 + 20\,000^2 + 20\,000 \cdot 3 = 800\,060\,000 \tag{6}$$

That is why it is more convenient to use different network architectures that can process large input data and extract important information from it using less number of parameters.

2.2.1 Convolutional network

Regular neural network transforms an input by passing it through a set of hidden layers to get a desired final output. *Convolutional neural networks* (CNN) are slightly different. First of all, they process data that has a known grid-like topology. Examples include image data that can be thought as a 2-D grid of pixels. Secondly, convolutional neural networks have two components.

- Feature extraction part
- The regression/classification part (it depends on a type of a concrete task)

In the first part of a network it performs *convolution* and *pooling* operations to detect features. In example, if there is an image of a human's face – then a network would recognize some parts of this face: ears, eyes, lips and etc.

In the second part of a network it uses extracted features to produce desired output by passing them through a set of hidden fully connected layers. This part is similar to the feedforward network architecture.

Convolution operation (fig. 2.6) is one of main blocks of any CNN. In math, a term *convolution* means a combination of two functions to produce a third function. In a case of CNN, the convolution is performed using *filter* (or *kernel*) which essentially is a matrix of weight coefficients. This operation is executed by sliding a filter over the input. A matrix multiplication is performed at every location and then a sum of all elements in the resulted matrix form a new pixel in the feature map. Usually, every convolutional layer has a set of n kernels to produce n feature maps on one input.

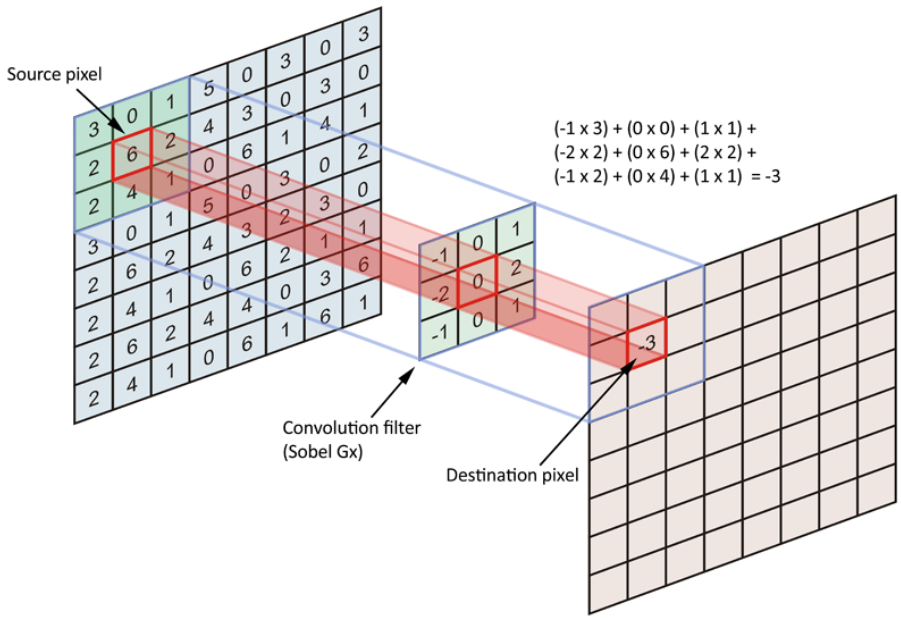


Figure 2.6: Visualization of the convolution operation.

An activation function is applied after the convolution operation to make output to be non-linear. In case of a CNN, it is usually *Rectified Linear Unit* (ReLU):

$$f(x) = \max(0, x) \tag{7}$$

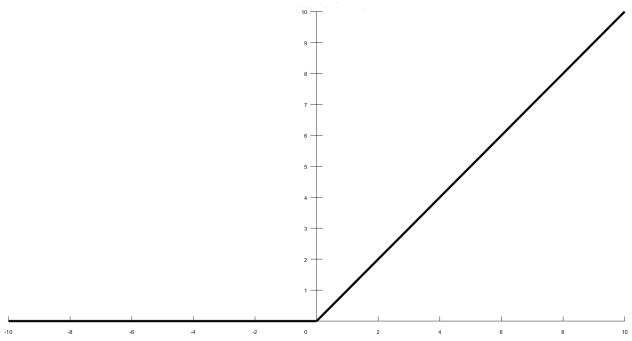


Figure 2.7: Rectified linear unit function.

According to *Krizhevsky et al.* [3]:

> Convolutional neural networks with very deep architecture train several times faster with ReLUs than their equivalents with hyperbolic tangent units.

Stride is a step size that the convolutional filter moves each time. It is usually 1. Because the feature map size is always smaller than an input size, it is needed to do something to prevent shrinking an image. This is where we use *padding* (fig. 2.8). We add pixels with zero value to surround the image with zeros, so that a feature map has same size as an input.

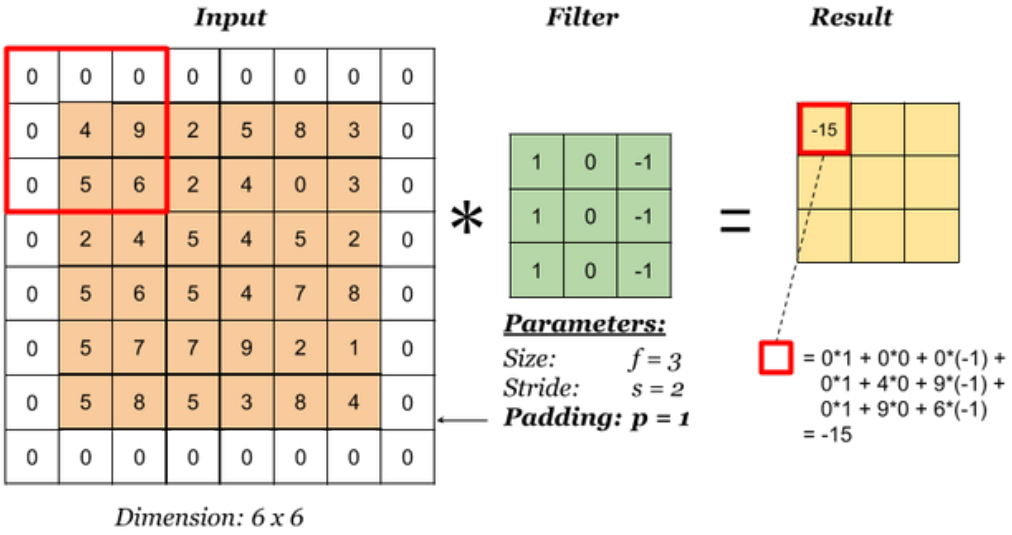


Figure 2.8: Padding visualization. In this case the feature map doesn't have the same size as an input, because a stride equals to 2.

It is a common practice to add the pooling layer (fig. 2.9) after the convolutional layer. The pooling layer also has a filter size, strides and performs *max* or *average* pooling operation by sliding a filter over the input. It significantly reduces the dimensionality of input data for hidden fully connected layers at the bottom of a network.

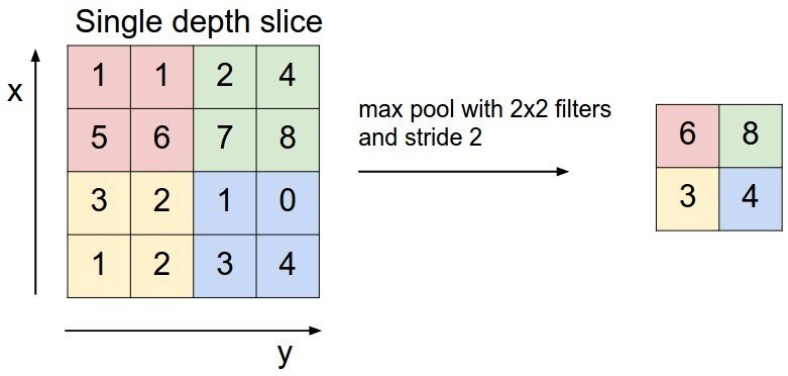


Figure 2.9: Example of the max pooling output on a 4x4 one-channeled image.

Convolutional neural networks implement two important ideas: *sparse interactions* and *parameter sharing*. Sparse interactions mean that we define kernel sizes to be smaller than the size of an input image. In example, an input image can contain thousands or millions of pixels, but we detect small features that have only tens or hundreds of pixels. Parameter sharing means that we reuse same parameter for more than one function. During the prediction step each parameter of regular neural network is used exactly once. In a case of a CNN, each element of a kernel matrix is used during computing a result on every input pixel. In conjuction with using of pooling layers that reduce the dimensionality of input data for hidden layers – it significantly decreases the number of parameters to train.

2.2.2 Regularization methods

Let's consider the figure 2.10. The training loss constantly decreases but at some point the evaluation loss starts to increase. In machine learning it is called *overfitting*. To prevent this we can apply a set of regularization techniques.

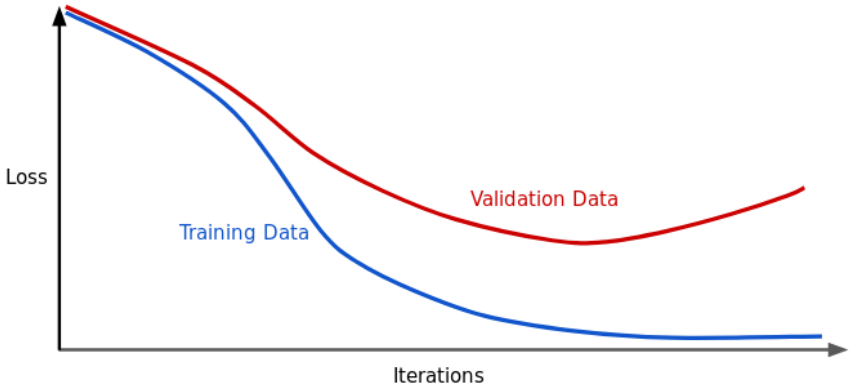
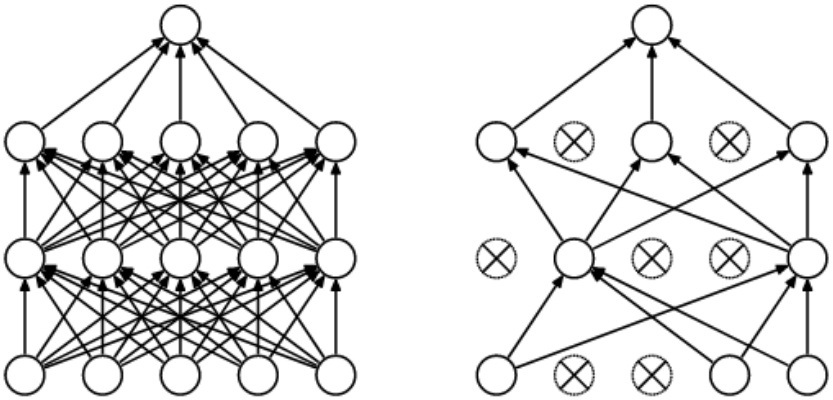


Figure 2.10: Example of overfitting the model.

Dropout Dropout consists in randomly setting unit's value to 0 at each update during training time. Each neuron has a probability p to be kept and probability $(1-p)$ to be dropped. The units that are kept are scaled by $\frac{1}{1-p}$, so that their sum is unchanged at

training time and inference time. Usually the dropout technique is only performed during training so that all units are kept during the evaluation step.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.11: Dropout operation.

According to *Krizhevsky et al.* [4], the activations of the hidden units become sparse with dropout operation, even when no sparsity inducing regularizers are present. Thus, dropout automatically leads to sparse representations. This improves generalization because it forces layers to learn the same "concept" with different neurons.

Local response normalization In a case of a CNN, ReLU units have a property that they do not need any kind of an input normalization. However, *Krizhevsky et al.* [3] developed a technique that aids generalization. Such method is called *Local Response Normalization*:

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^{\beta} \tag{8}$$

where $b^i_{x,y}$ is the response-normalized activity, $a^i_{x,y}$ is the activity of a neuron computed by applying kernel i at position (x, y) and then applying ReLU nonlinearity. Sum runs over n adjacent kernel maps at the same spatial position, and N is the total number of kernels in the layer.

Such lateral inhibition is observed in the brain, and we can also think of it as helping sharpening the response.

L2 regularization Instead of aiming on mizimizing just the loss function value:

$$\theta = \arg\min_{\theta}(\text{loss}(x, \theta)) \tag{9}$$

we will now minimize loss + complexity of the model:

$$\theta = \arg\min_{\theta}(\text{loss}(x, \theta) + \text{complexity}(\theta)) \tag{10}$$

In terms of *L2 regularization* the complexity of the model is given by:

$$\text{complexity}(\theta) = \frac{\alpha}{2} \cdot \|w\|_2^2 = \frac{\alpha}{2} \cdot (w_1^2 + w_2^2 + ...) \tag{11}$$

where α is a scale factor in a range of $[0; 1]$ and w is a vector of all weight coefficients of the neural network.

This method prevents coefficients to fit so perfectly to overfit. Scale factor is often choosed empirically and can be thought as a hyperparameter of a network.

2.3 Projection method [MD]

Several methods for projecting spherical surfaces on to a plane have been invented over the years espacially in order to create maps for navigation. A broad summary of map projections including their features and flaws can be found on [5, 6].

In the scope of this project we decided to use an equal-area type of projection named *Mollweide-projection*. The decision was based on the plan of using the convolutional neural network also for energy reconstruction. The transformation is performed with the following set of equations:

$$x = 2\sqrt{2} \cdot \frac{\lambda}{\pi} \cdot \cos\theta \cdot R \,, \tag{12}$$

$$y = \sqrt{2} \cdot \sin\theta \cdot R \,, \tag{13}$$

where λ is the longitudinal, θ is the azimuthal angle and R is the radius. Since the radius is constant over the surface of the sphere, it is basically used as a scaling factor. Since the positions of the PMTs are given in cartesian coordinates, they have to be converted to spherical using standard equations.

An example of event data plotted using the projection method is show in fig. 2.12. Figure 2.13 shows the same event but rotated in order to show the charge center in the center of the projection. This rotated image also clarifies the non-homogenic positioning of PMTs in the azimuthal angle.

In order to feed the data to the neural network, the projections will be interpolated to smaller images with fixed dimension for charge and first hit-time information. The charge centered images will also be tested as input for the convolutional network in order to add this meta information to the prediction.

For additional tests also the sinusoidal projection may be of interest since it preserves distances. This may be of importance for angular vertex prediction using the convolutional neural network.

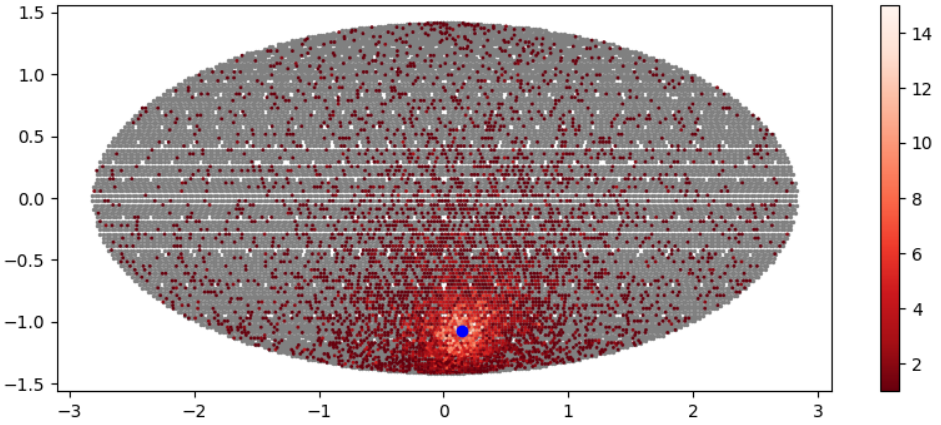


Figure 2.12: Scatter plot of PMT charge data in Mollweide projection. Grey dots represent PMTs which did not acquire a signal at all. The strength of signals is projected with a red-shade colormap as illustrated in the colorbar. The blue dot represents the vertex position determined using the charge center method (only angular information obviously).

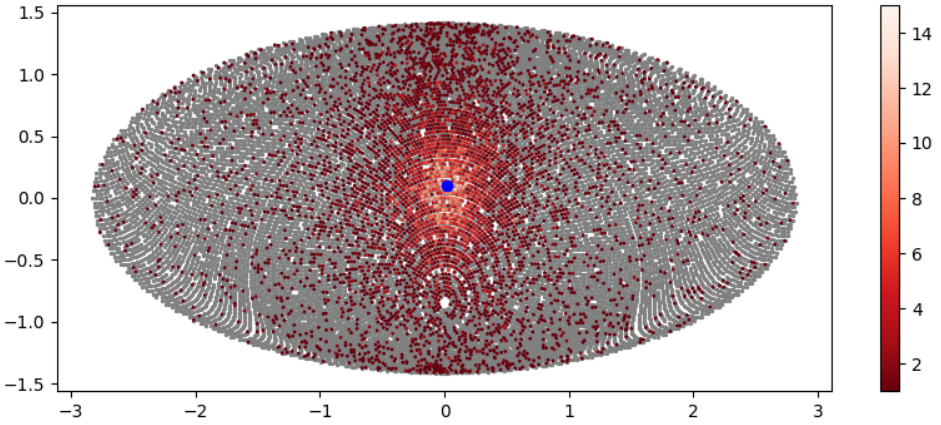


Figure 2.13: Same projection method but the coordinate system was rotated before projection to move the charge center to the middle of the projection.

3 Vertex reconstruction experiments

The results of the two approaches for vertex reconstruction using neural networks are discussed in the following chapters. Feed forward network essentially does correction of the charge center estimation based on summary data. Convolutional neural network produce the vertex reconstruction based on charge and time data of individual PMTs.

3.1 Feed forward network [MD]

The *feed forward network* (FFN) is a comparably (to the later discussed convolutional neural network) simple neural network. In the following the input data, architecture and results are presented.

3.1.1 Input data

The input data of our FFN consists of five or eight values per event:

- The total number of photoelectrons of all PMTs ($N_{p.e.}$),
- the mean first hit time ($\bar{t}_{hit}$),
- and the three cartesian and/or spherical coordinates of the estimated vertex position using the charge center method (see section 2.1.1).

If both coordinate representations of the charge center are used, the input vector consists of eight values, five otherwise.

Preparing the input data for the neural network from simulation files (ROOT-files) is done via `FFNdata.py`-script, converting raw data of each PMT's photo electrons (p.e.) and first hit time (t_{hit}) to the meta data described above and saved to hdf5-files.

3.1.2 Architecture

The basic structure of a neural network is described in the theory chapter 2.2. The used FFN consists solely of the input and output layer and multiple hidden layers. The main structure of the neural network is shown in fig. 3.1.

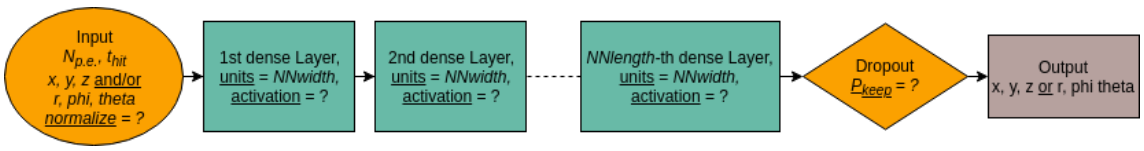


Figure 3.1: The general architecture of the feed forward network. Some of the hyperparameters which have to be optimized are indicated (underlined) and the dashed line indicates the variable number of dense layers.

The architecture design parameters like number of neurons in a hidden layer and number of hidden layers and type of input data where varied to find the optimal design. Additionally the typical hyperparameters of the neural network have to be varied to achieve

Parameter	Description	Value Range
NNwidth	size of dense layer	100 to 8192
NNlength	number of dense layers	2 to 10
activation	type of activation function	*ReLU*, *tanh*
batch size	number of events per batch	16 to 8192
optimizer	type of optimizer	GradDesc/Adam
start-LR	start value of the learning rate	0.000 05 to 0.1
LR-decay[1]	decay rate of learning rate	0.90 to 0.999
dropout	keep-rate of the drop out layer	0.01 to 0.99
normalize	normalization of input data	true/false
cartesian	add charge center in cartesian coord. to input	true/false
spherical	add charge center in spherical coord. to input	true/false
sph-label	use spherical coordinates for the label data	true/false

Table 1: List of hyperparameters and architecture design options. These parameters need to be optimized to increase the prediction accuracy of the neural network.

highest precision and accuracy in the prediction. All parameters which have been varied are listed in tab. 1 included the tested ranges.

Since the output of the neural network is choosen to be the three coordinates of the vertex position, the mean euclidean distance is choosen in case of cartesian coordinates:

$$\overline{\Delta R} = \frac{1}{n} \sum_i |y_i^{true} - y_i^{pred}| \,, \tag{14}$$

with n being the size of the batch, i the index of the event and y_i^{true}/y_i^{pred} the real vertex position and the predicted one respectivly. In case of spherical labels the *MSE* (eq. 4) is used. Both loss functions are equally performing in the sense that they both can be used for optimizing the prediction. The euclidean distance is just more convinient in case of cartesian coordinates.

For initiallization of the kernel (weights and biases) the default *glorot uniform initializer* is used which draws samples from a uniform distribution. Additionally the samples inside each batch are shuffled.

3.1.3 Results

Optimizing the networks parameters was done by training and evaluating multiple instances of the network with varying parameters in nested loops. In order to minimize the time for this optimization only subsets of parameters have been varied for each optimization attempt and the best result is used for further optimization attempts using a different subset of parameters.

In the beginning of the optimization process the input data was normalized by default and activation function for the dense layers was fixed to *tanh* due to exploding weights. Later these parameters were changed too, since the effect did vanish with better parameters (especially with use of *Adam Optimizer*).

The best results were archieved with the architecture shown in fig. 3.2 using the Adam optimizer, normalized input data including cartesian and spherical coordinates of the

charge center and the *ReLU* activation function.

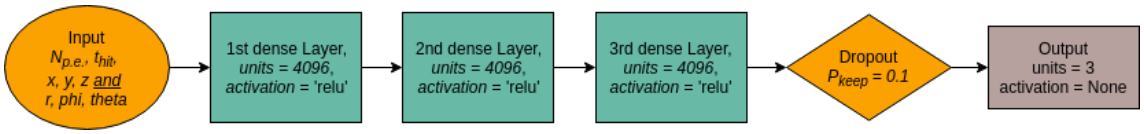


Figure 3.2: The simple architecture of the feed forward network. The shown architecture is the currently best performing one. More details on other tested *dimensions* (number of dense layers and units/neurons in each layer) and hyperparameters are given in the text.

The networks geometry was optimal with a low number of big dense layers. The *Start-Learning Rate* (start-LR) was optimal at 3×10^{-4} and the batch size could be choosen as big as 8192 without any change in the result but reducing the computational effort. Using spherical coordinates for the label results always in very bad results even with otherwise optimal parameters.

The evaluation results of the FFN are shown in fig. 3.3. The correlation between the real radius and the predicted one, which have been present in the charge center reconstruction, is eliminated. ΔR over r^3 reassambles a uniform distribution in r^3 and poisson in ΔR. The mean error on the euclidian distance is about $25 \pm 12\,\mathrm{cm}$. Important to mention though: The results include outliers with values of $\Delta R > 200\,\mathrm{cm}$ which arise due to outlines in the charge center data and therefore are impossible to correct by the neural network.

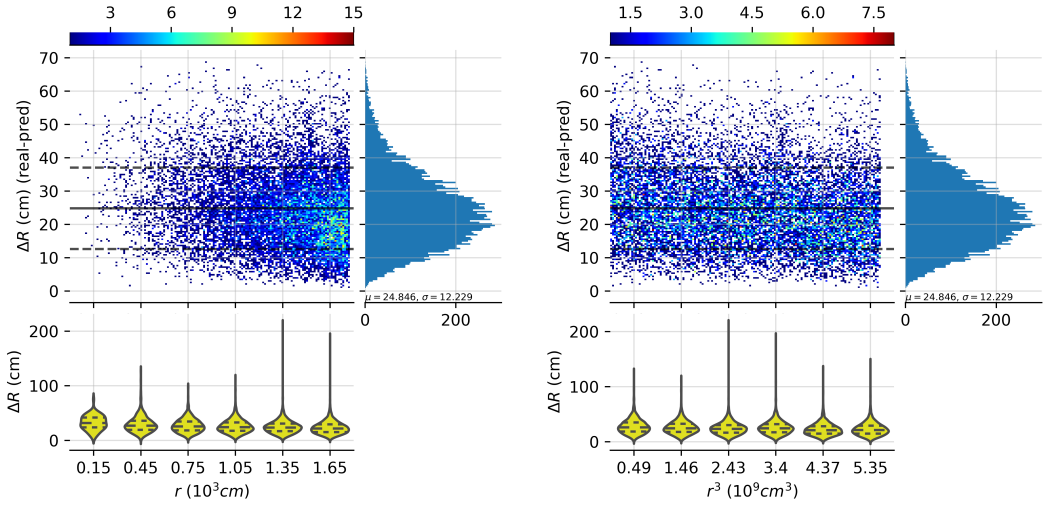


Figure 3.3: Results of vertex reconstruction with the feed forward network. The training data set consisted of 140 000 5 MeV-events of Positron and the evaluation set of 16 000 events. Shown is the absolute distance between reconstructed and real vertex position over radius and cubic radius. The respective line-plots can be found in appendix A.2.

Investigating the results in single coordinates (see fig. 3.4) shows far less structure in the cartesian coordinates, but still some depencies and offsets are visible. The prediction results plotted in spherical coordinate show a deviation in the radius of about $-1.5 \pm 16.0\,\mathrm{cm}$. The deviations of the angles θ and ϕ are about $-0.005 \pm 0.020\,\mathrm{rad}$ and $0.00 \pm 0.05\,\mathrm{rad}$

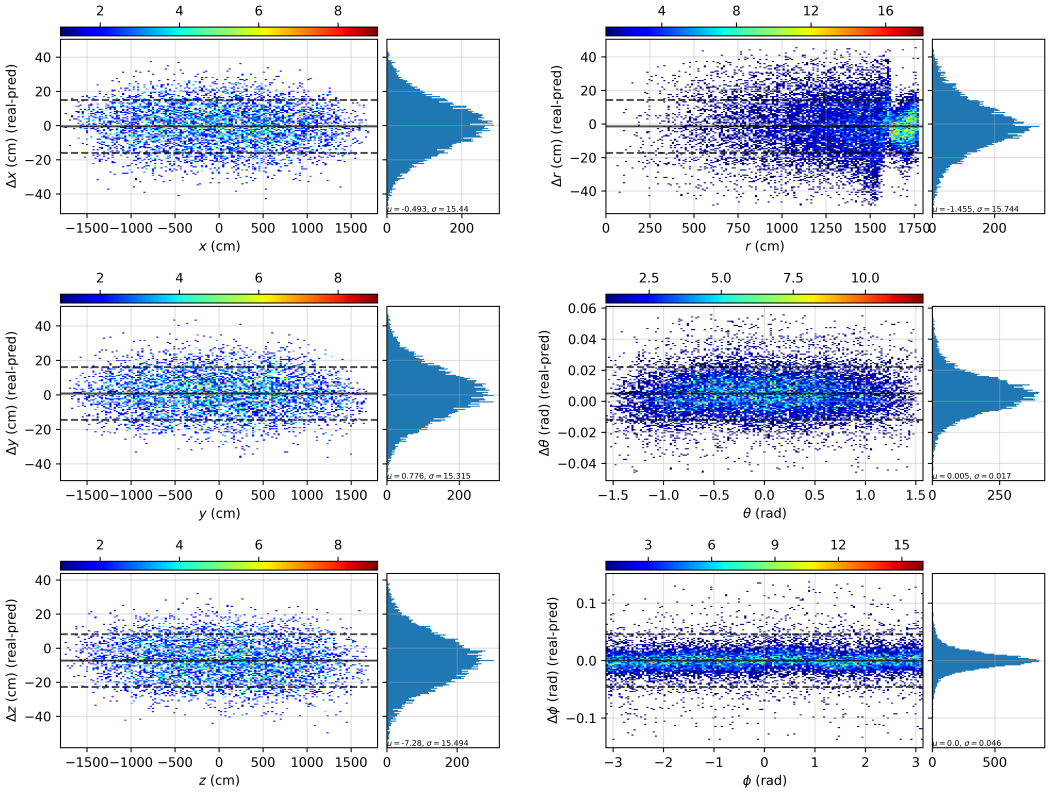


Figure 3.4: Results of vertex reconstruction with the feed forward network. The training data set consisted of 140 000 5 MeV-events of Positron and the evaluation set of 16 000 events. Shown are deviations in cartesian and spherical coordinates of the reconstructed vertex position from the real vertex position. The respective line-plots can be found in appendix A.1.

respectivly. These deviations seem small, but for radii of e.g. 16 m the error in the angle corresponds to up to 80 cm.

In order to investigate this further the individual coordinates are also plotted over the real cubic radius in fig. 3.5. The results show a decreasing spread of the angular deviation with increasing radius. Therefore the above described effect is deminished, but still the angular deviation is now dominant over the radial one. This may be due to the abscence of structure in the charge center angle information.

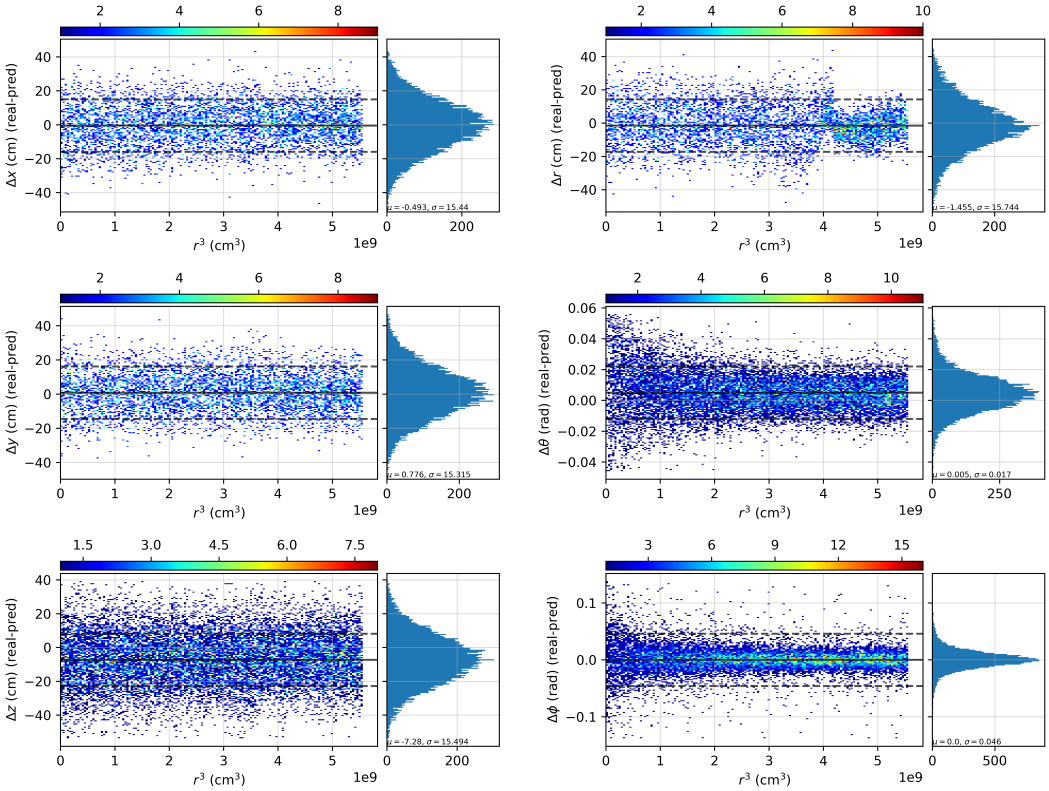


Figure 3.5: Results of vertex reconstruction with the feed forward network. The training data set consisted of 140 000 5 MeV-events of Positron and the evaluation set of 16 000 events. Shown are deviations in cartesian and spherical coordinates of the reconstructed vertex position from the real vertex position **over the real cubic radius**. The respective line-plots can be found in appendix A.1.

3.2 Convolutional network [DS]

Tensorflow [7] was choosed as a core framework for building the neural network model. It has many powerful tools, provides high performance and allows models to be saved at checkpoints and loaded into any Tensorflow instance.

3.2.1 Input data

The inputs are 2-channeled images with resolution 192x96, where the first channel is charge data and the second channel – time data. All $\approx 20\,000$ PMT's are arranged into rectangular images using the Mollweide projection method. No additional input normalization is performed, since ReLU units has a property that they do not require it [3].

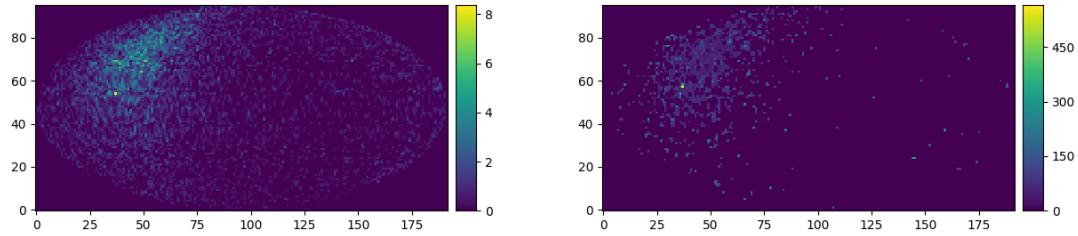


Figure 3.6: Rectangular input images of one single event. Shown is charge data (left) and time data (right). Y-axis is θ angle and X-axis is ϕ angle. This pictures are arranged into one 2-channeled image and are used for feeding the network.

All images are stored as a NumPy arrays in HDF5 files. Tensorflow provides huge dataset API for defining custom input pipeline architecture. We extract data from HDF5 files using multiple CPU cores and prefetch every next batch on CPU while current batch is processed on GPU.

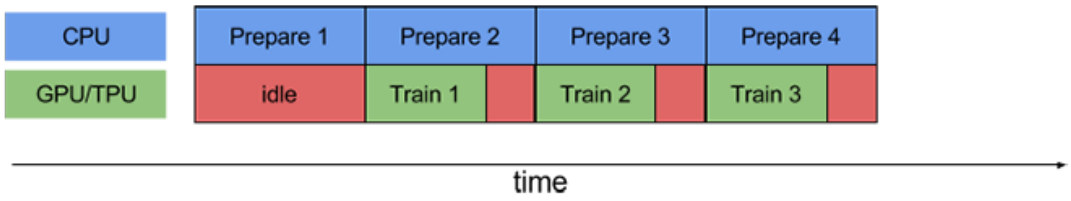


Figure 3.7: Device idle time visualization.

3.2.2 Architecture

The network has 1 input layer, 4 convolutional layers, 4 pooling layers, 2 fully connected layers and 1 output layer in total. We applied local response normalization layer after each

convolutional one to aid the generalization. The coefficients for LRN are: $\alpha = 0.001/9.0$, $\beta = 0.75$, $k = 0.5$. The dropout is performed before the output layer to prevent overfitting. In this way, the architecture looks like:

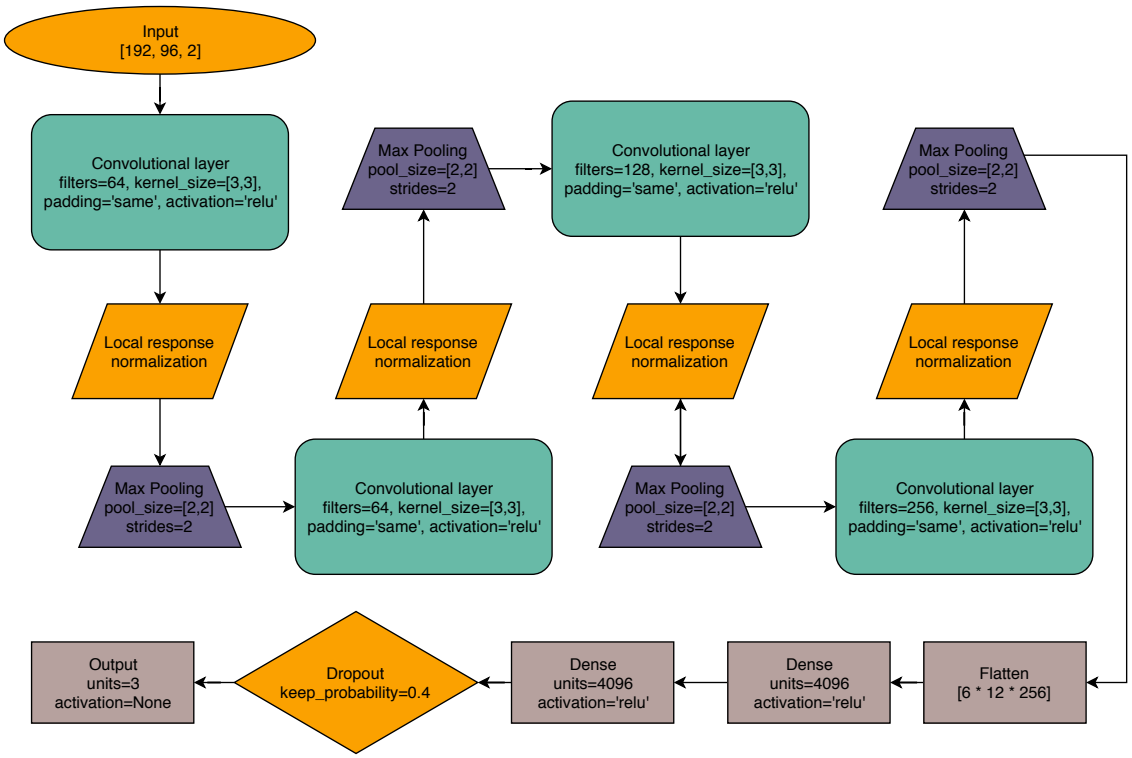


Figure 3.8: Network architecture.

Adam optimizer [8] was found out as a good optimization method for the vertex reconstruction task. This algorithm is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The update rule for parameter θ with gradient g is described at the section 2 of the paper [8]:

$$
\begin{aligned}
t &= t + 1 \\
\mathrm{lr}_t &= \alpha \cdot \sqrt{\frac{1 - \beta_2^t}{1 - \beta_1^2}} \\
m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g \\
v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g \cdot g \\
\theta_t &= \theta_{t-1} - \mathrm{lr}_t \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}
\end{aligned}
$$

where t is the number of current iteration, lr_t is the learning step size at iteration t, α is the initial learning step size, ϵ is the constant for numerical stability and for prevention division by zero , β_1 and β_2 are constants in range $(0; 1)$, usually they are choosed close to 1. We use default settings proposed by authors of the paper [8]: $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. For the initial learning step size α we found out that the value $3 \cdot 10^{-4}$ is best.

The *Mean Squared Error* function is used as a loss function. We add the L2 regularization term to it as it is decribed in 2.2.2. The scale factors for weight coefficients are

0.01 and 0.05 for convolutional and fully connected layers respectively.

The weight initialization method is significantly important in deep neural network architectures. By using Xavier weight initialization [9], we make sure that the weights are not too small but not too big to propagate accurately the signals. With each passing layer the variance remains same. It helps to keep the signal from exploding to a high value or vanishing to zero.

We trained our model with the batch size of 140 events, the initial learning rate of $3 \cdot 10^{-4}$ and a number of epochs of 64. Additional epochs do not improve evaluation much but require a lot of computational time to proceed.

3.2.3 Results

Our results are summarized in fig. 3.9 and fig. 3.10. The evaluation set consists of 16 000 events. The error in total distance has mean of $\approx 23\,\text{cm}$ and the standard deviation of $\approx 11.2\,\text{cm}$. The results doesn't show a rising error with rising vertex radius as it is in the charge center method, but there are more outliers at high radii. A possible reason for this may be that the CNN learned patterns about events with high radius, but some of events are still hard to predict. We found that the larger size of a training set helps to fight outliers.

The main disadvantage of using CNN is that it can take large amount of time to find a good set of hyperparameters in case of a change of the network's architecture.

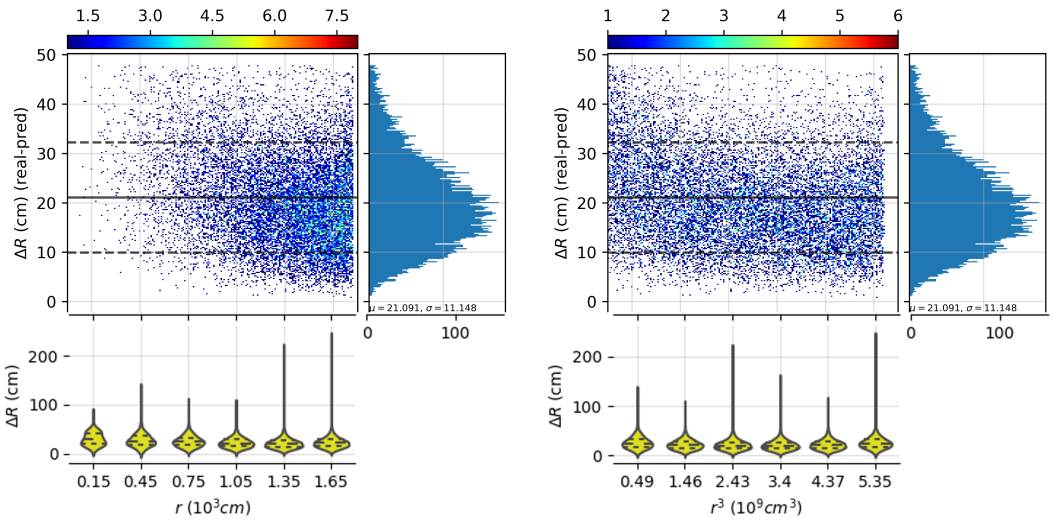


Figure 3.9: Results of vertex reconstruction with the convolutional network. The training data set consisted of 140 000 5 MeV-events of Positron and the evaluation set of 16 000 events. Shown is the absolute distance between reconstructed and real vertex position over radius and cubic radius. The straight black line shows μ value, the top and bottom dotted lines show $\mu + \sigma$ and μ - σ values respectively.
The bottom plots are violin plots. The outer shape represents all presented values, with a thickness representing how often they occur. The thickest section represents the mode. The next layer inside represents the values that occur 95% of the time. The quartiles are drawn as a black lines inside each violin.
The respective line-plots can be found in appendix A.9.

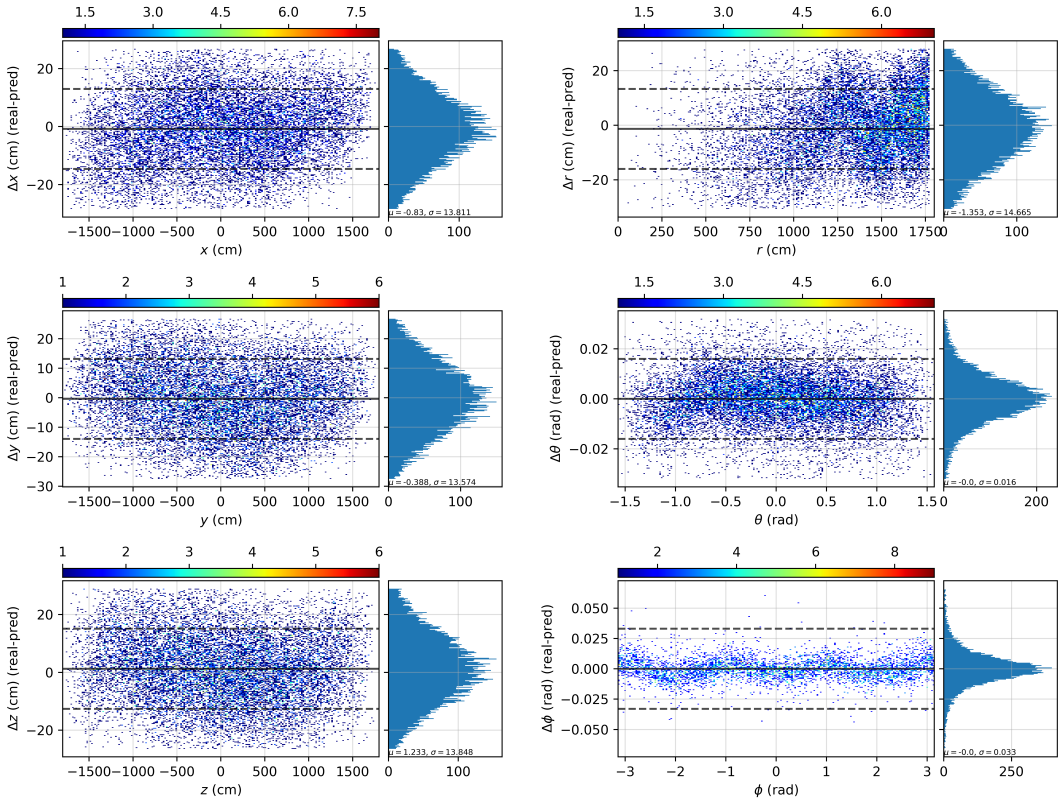


Figure 3.10: Results of vertex reconstruction with the convolutional network. The training data set consisted of 140 000 5 MeV-events of Positron and the evaluation set of 16 000 events. Shown are deviations in cartesian and spherical coordinates of the reconstructed vertex position from the real vertex position. The straight black line shows μ value, the top and bottom dotted lines show $\mu + \sigma$ and μ - σ values respectively.
The respective line-plots can be found in appendix A.8

4 Conclusion & Outlook

4.1 Summary

The goal of this work was to develop and describe a neural network for predicting the primary vertex position in the JUNO experiment. Two networks for two different tasks were succesfully designed and tested. Our results show that such methods have high potential and they are comparable to classical methods. However, there are a lot of things to research and a lot of work to be done to reach the minimal required precision of 10 cm.

4.2 Outlook

Vertex / track reconstruction All results in previous section are produced on Positron events. Future plans include the vertex reconstruction on electron events and track reconstruction on muon events.

Event classification In addition to vertex and energy reconstruction there is also the task of an event classification. It is important to classify a type of a particle depending on charge and time data of all PMT's. Convolutional networks are especially good in terms of image classification so this instrument could give satisfactory accuracy.

Different input data There are a lot of possibilities of using several projection methods to arrange PMT's into rectangular images. The resolution of input images also has an influnce on total distance error and computational time. In the case of a feedforward network we can add multipole data of charge center information as an additional input variables.

Search for better hyperparameters The number of hyperparameters leads to a substantial number of choices to be made when creating a neural network, and that these choices will affect the success and failure of the model.

Unfortunately, it takes much time to find a good set of hyperparameters with using deep and complex architectures, since the number of choices is large, and one training process requires a lot of computational time.

The usage of modern algorithms for hyperparameter optimization (like [10]) may improove this situation.

Investigating more energies Both of convolutional and feedforward networks are trained on 5 MeV Positron events. To get more complete results it may be needed to observe different energies, in example in range from 1 to 10 MeV.

Modification of the input pipeline for multiple GPUs Tensorflow has Distribution Strategy API which provides a way to distribute training across multiple devices. Currently, it supports only one type of strategy, called Mirrored Strategy. It does in-graph replication with synchronous training on many GPUs on one machine. Essentially, it creates copies of all variables in the model's layers on each device. Then all-reduce operation

is performed to combine gradients across the devices before applying them to the variables to keep them in sync.

To apply Mirrored Strategy on a model, it is needed to modify the input pipeline architecture. The extraction and prefetching of the data on CPUs should be syncronised and distributed correctly with respect to multiple models.

Fixing weight decay in Adam Weight decay is commonly used in convolutional neural networks and decays the weights θ_t after every parameter update by multiplying them by a decay rate α that is slightly less than 1:

$$\theta_{t+1} = \alpha \cdot \theta_t \tag{15}$$

When we use gradient descent as an optimization method, weight decay can also be understood as L2 regularization term. But with using Adam it doesn't equal to L2 regularization, since the gradient is modified in both the momentum and Adam update equations. *Loshchilov and Hutter* [11] proposed a way to fix it by adding weight decay after the parameter update in the original definition. Then the parameter update rule looks like the following:

$$\theta_t = \theta_{t-1} - \mathrm{lr}_t \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} - \mathrm{lr}_t \cdot \alpha \cdot \theta_{t-1} \tag{16}$$

Fixing exponential moving average in Adam *Reddi et al.* [12] state that the exponential moving average of past squared gradients is another reason for the poor generalization of adaptive learning rate methods. It has been found out that some minibatches provide large and informative gradients, but it happens rarely in complex tasks and the exponential averaging diminishes their influence. Thus it leads to poor convegence.

The authors propose a new algorithm, which is called AMSGrad, that uses the maximum of past squared gradients rather than the exponential average to update the parameters. The AMSGrad update rule can be seen below:

$$\begin{aligned}
\hat{v}_{t-1} &= \frac{v_{t-1}}{1 - \beta_2^t} \\
m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g \\
v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g \cdot g \\
\hat{v}_t &= \max(\hat{v}_{t-1}, v_t) \\
\theta_t &= \theta_{t-1} - \mathrm{lr}_t \cdot \frac{m_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}$$

A Appendix

A.1 Additional plots for the feedforward network

Plots of deviation in predicted and true positions over the radius and cubic radius for results of the feedforward network are shown in the paragraph below.

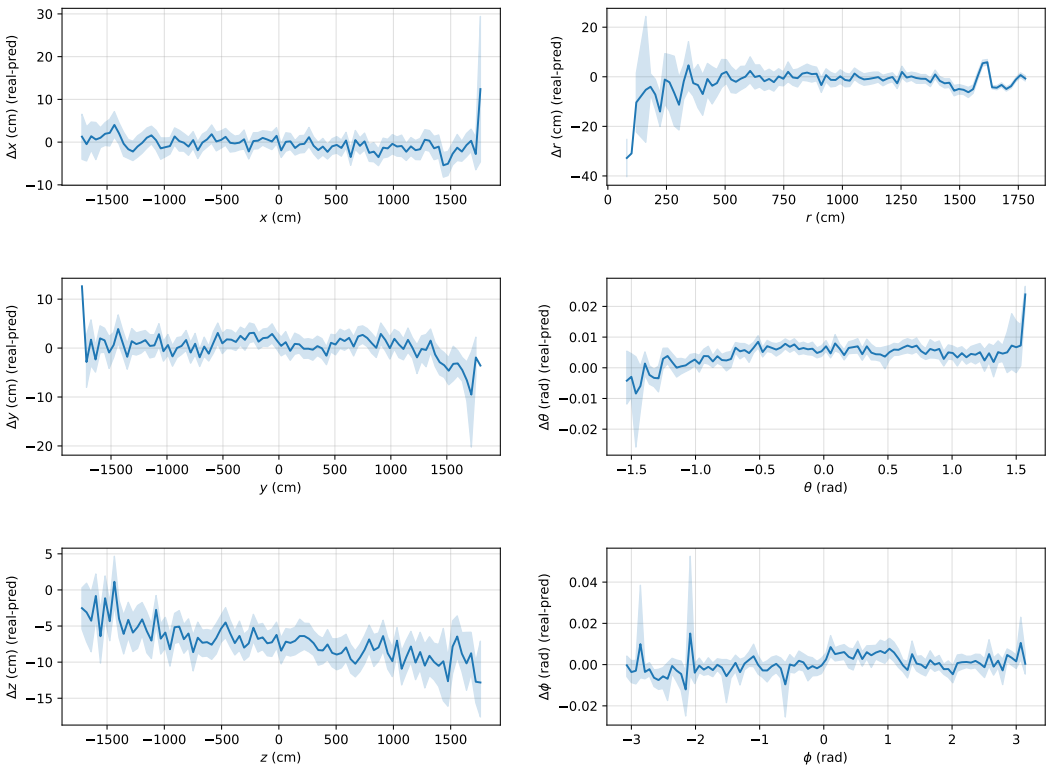


Figure A.1: Deviation of the predicted position from the real position in each individual coordinate. The band around the line plot indicates the 1σ region.

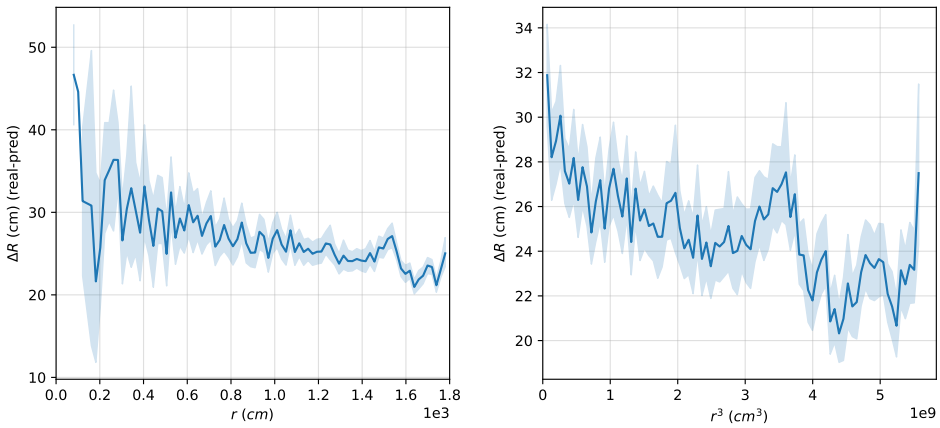


Figure A.2: Euclidian distance of the predicted position from the real position. The band around the line plot indicates the 1σ region.

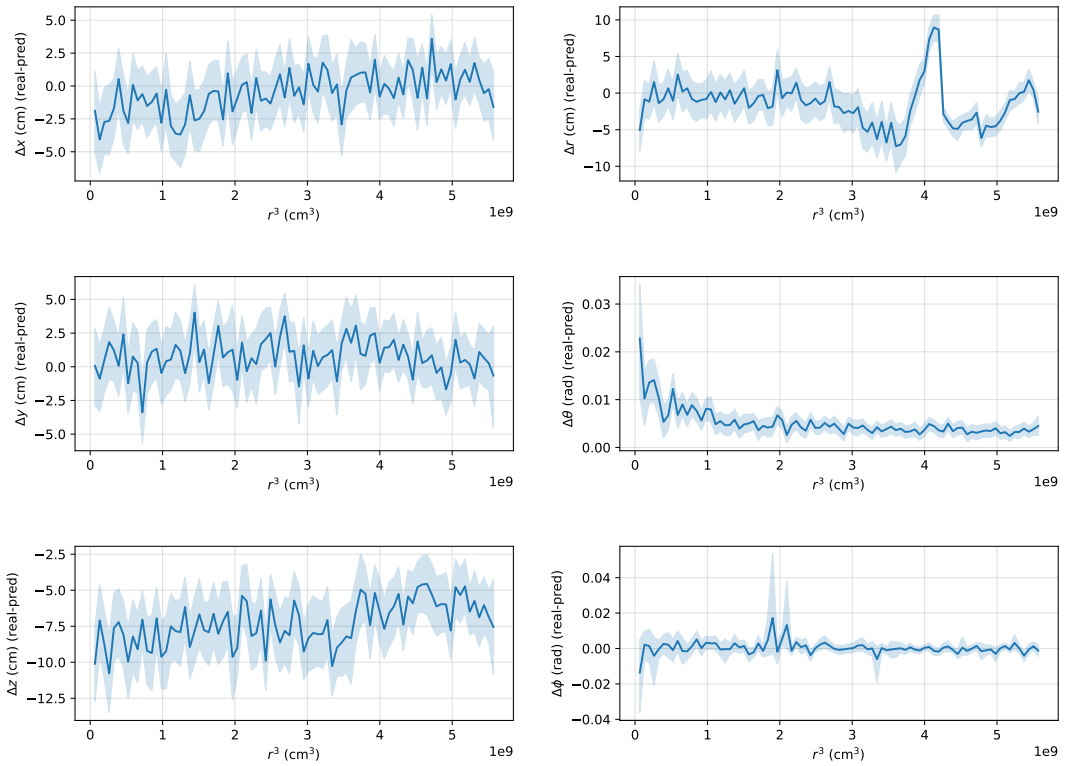


Figure A.3: Deviation of the predicted position from the real position in each individual coordinate plotted over the real cubic radius. The band around the line plot indicates the 1σ region

A.2 Example of inputs for the convolutional network

Projections of charge and time data which are used as an input for the convolutional network are shown in the paragraph below.

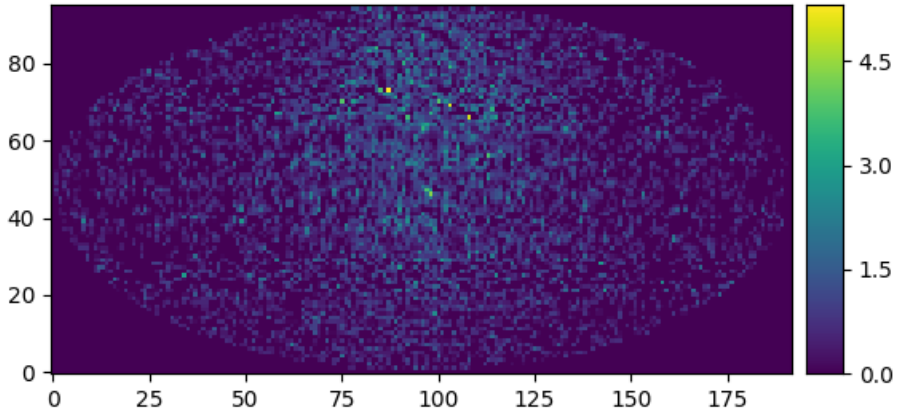


Figure A.4: Example of input for the CNN.

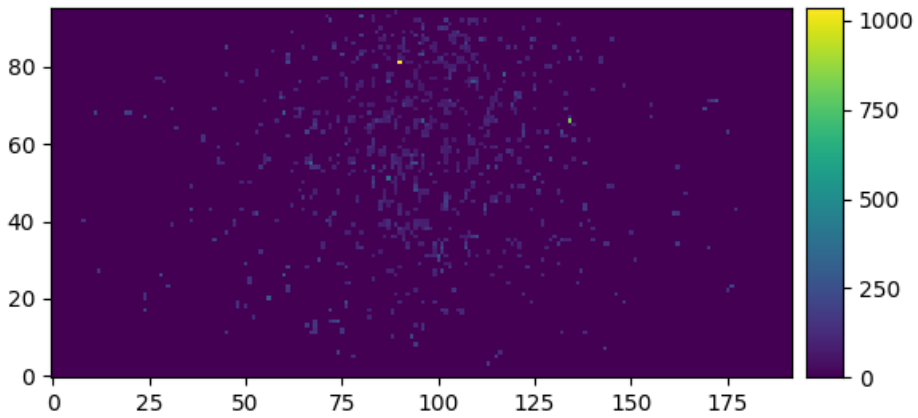


Figure A.5: Example of input for the CNN.

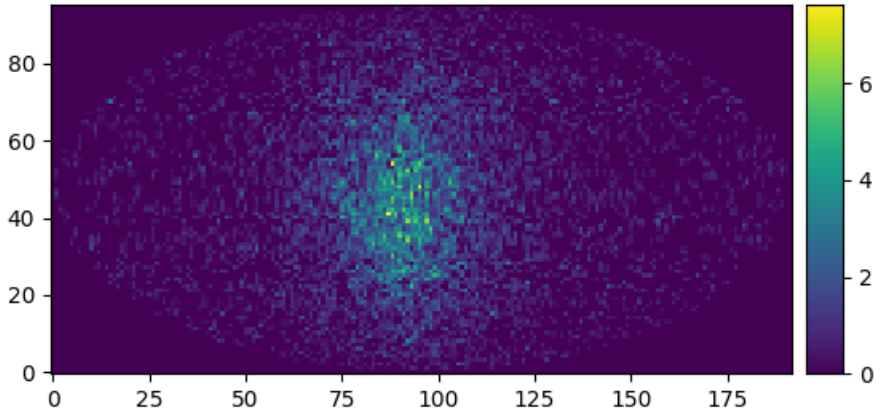


Figure A.6: Example of input for the CNN.

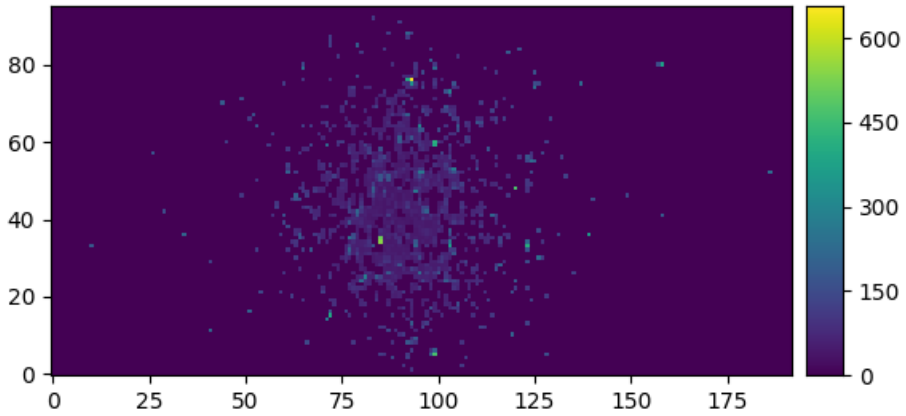


Figure A.7: Example of input for the CNN.

A.3 Additional plots for the convolutional network

Plots of deviation in predicted and true positions over the radius and cubic radius for results of the convolutional network are shown in the paragraph below.

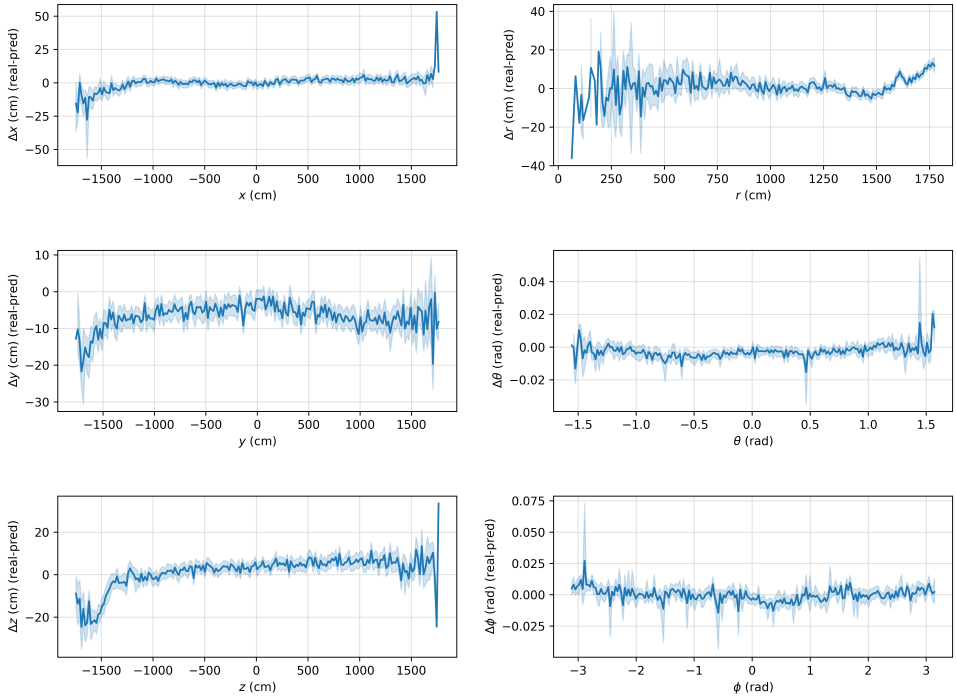


Figure A.8: Euclidian distance of the predicted position in CNN results from the real position. The band around the line plot indicates the 1σ region.

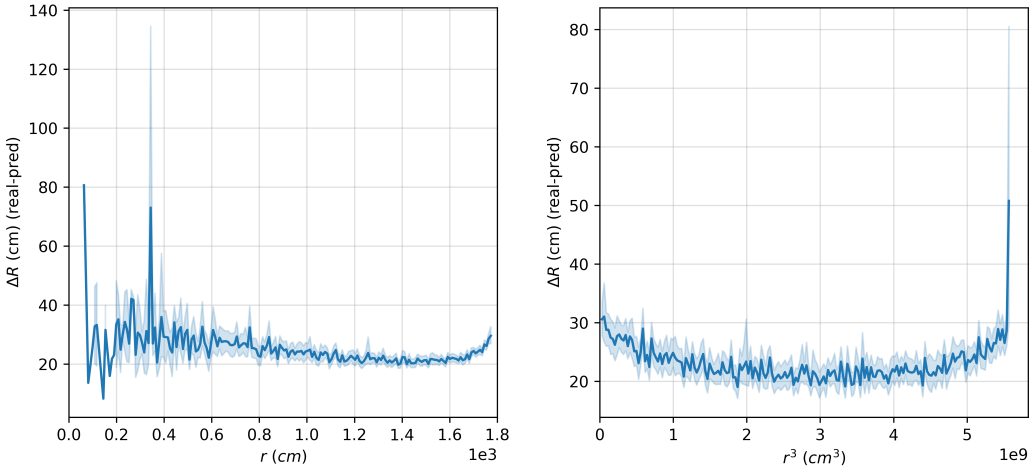


Figure A.9: Deviation of the predicted position in CNN results from the real position in each individual coordinate plotted over the real cubic radius. The band around the line plot indicates the 1σ region

B Bibliography

References

[1] Q. Liu, M. He, X. Ding, W. Li, and H. Peng. A vertex reconstruction algorithm in the central detector of JUNO. ArXiv e-prints, March 2018.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems 25, pages 1097–1105. Curran Associates, Inc., 2012.

[4] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15:1929–1958, 2014.

[5] Carlos A. Furuti. Cartographical Map Projections : Cartography, the science of map-making, comprises many problems and techniques, 2018. [Online; accessed 05-July-2018].

[6] Wikipedia contributors. List of map projections — Wikipedia, the free encyclopedia, 2018. [Online; accessed 28-June-2018].

[7] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[8] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. CoRR, abs/1412.6980, 2014.

[9] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, volume 9 of Proceedings of Machine Learning Research, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[10] Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In Proceedings of the 34th International Conference on Machine Learning, volume 70 of Proceedings of Machine Learning Research, pages 1165–1173. PMLR, 2017.

[11] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. CoRR, abs/1711.05101, 2017.

[12] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In International Conference on Learning Representations, 2018.

C Acknowledgements

First of all we want to thank Dimitry Naumov for giving us the opportunity to work in this amazing group at the Dzelepov laboratory of nuclear problems and also for his motivating and inspiring feedback while working on this project.

Many thanks to Maxim Gonchar for being our supervisor, supporting us with everything we needed for conducting our experiments and for discussing possibilities and problems even on his vacation!

Also special thanks to Konstantin Treskov for amazing IT and tea support ;)

Thanks to JINR Summer Student program coordinators Elena Karpova and Elizabeth Budennaya for doing a flawless job preparing the program, conducting lab tours and helping students in all situations.

Thanks to HybriLIT team for giving an access to the heterogeneous computing system.

Thanks to SPbSU-cluster and JINR Cloud teams for providing an access to virtual machines.