Helmut Hörner

An Exact, Fast Algorithm for the "Charge Refinement Problem" in the Simulation of Heavy-Ion Collisions, in Comparison to Slimmed-Down Neural Networks

Project Report
Version 3

Vienna University of Technology
Institute of Theoretical Physics

Vienna, November 2019

Abstract

In certain numerical simulations of the early stages of heavy-ion collisions it is necessary to split charges in Wigner-Seitz cells into smaller sub-charges, which are then smoothly distributed as to approximate a continuous charge distribution. "Smooth" means that the discrete fourth derivate should be constant within each Wigner-Seitz cell. In this paper, we demonstrate that simple, dense neural networks can be trained to learn this charge distribution task, and how these networks reveal the linearity of the problem.

We derive a very fast linear algorithm for directly calculating the exact charge distribution without neural networks, and present a C++ implementation of this algorithm. Eventually, we go back to neural networks and present more refined convolutional network architectures with a significantly reduced number of trainable parameters. Because of their lean structure, these refined networks are a good alternative to the exact solution when it comes to a very large number of charges.

Contents

1 Introduction

In [Gelfand et al, 2016], a numerical simulation of the early stages of heavy-ion collisions in 3+1 dimensions is presented. It makes use of the nearest-grid-point (NGP) interpolation method [Moore et al, 1998], where a particle charge $Q(t)$ at a specific time t is fully mapped to the closest lattice point. The charge density only changes when a particle crosses the boundary in the middle of a cell such that its nearest-grid-point changes. These boundaries can be formally defined with a Wigner-Seitz lattice, with lattice points marking the center of each cell.

However, for the simulation not to produce a lot of numerical artifacts, it is crucial to approximate a continuous charge distribution. Therefore, it is not sufficient to represent the total charge per Wigner-Seitz cell with a single charge. Instead, in each Wigner-Seitz cell the total charge must be split into smaller sub-charges, which are then smoothly distributed as to approximate a continuous charge distribution (see Figure 1).

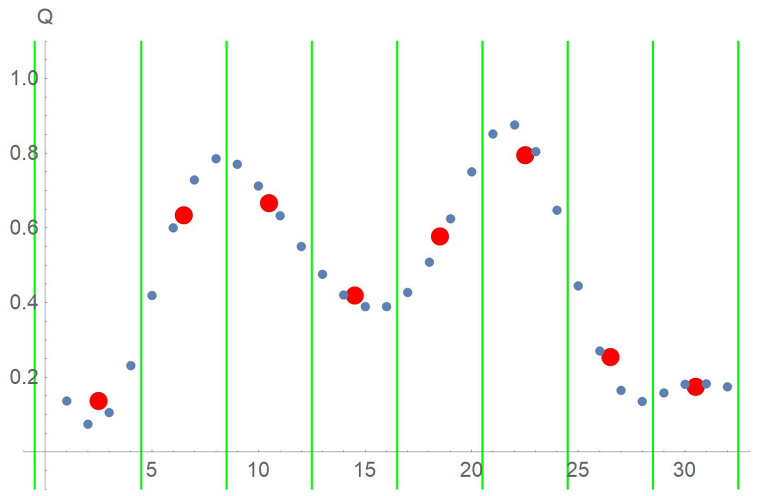

Figure 1: Originally, the total charge in each Wigner-Seitz cell (green lines) is represented by a single charge (red dots). By splitting these single charges into multiple charges (blue dots), and distributing them so that the discrete fourth derivative is constant within each cell, a continuous charge distribution can be approximated. Please note that the refined charges (blue dots) are plotted with four times their actual values to better set them in visual context with the original charges (red dots).

In this context, "smooth" means that the discrete fourth derivate should be constant within each cell. In [Gelfand et al, 2016], a (rather slow) iterative algorithm was implemented to simulate such a charge distribution.

The initial goal of this project was to find out to what extent a deep-learning neural network could be trained to speed up this charge distribution task, and (if possible) to provide such a trained network.

As we will show in the following chapter 2, it turned out that

(i) Deep networks can be trained to fulfill this task;

(ii) the fewer layers a network has, the better it works for this job;

(iii) so that, eventually, not-at-all-deep dense networks with no hidden layers work best;

(iv) and, additionally, such simple networks provide the best results if any non-linear activation function is abandoned in favor of a simple linear activation function.

Such an extremely simplified neural network can (because of its linear activation function) be represented by a simple linear vector/matrix equation. Therefore, the problem can obviously be reduced to linear algebra. Consequently, we derived the exact linear solution, which we present later on in chapter 3.2, and a corresponding C++ implementation in chapter 3.3.

Eventually, in chapter 4 we compare the weight matrix of a trained dense neural network with the exact solution, and further demonstrate how it still makes sense to use a convolutional network to quickly solve the charge-distribution problem for large charge distributions.

2 The Deep Learning Approach

2.1 Generating Training Data with the Original Iterative Algorithm

2.1.1 Theory

This chapter gives an overview of the iterative algorithm as presented in [Gelfand et al, 2016], that has been re-implemented by us in Python in order to generate the training data set.

(1) Let Q_j be the total charge in the j-th Wigner-Seitz cell. In this first step we create N sub-charges q_i for each original charge Q_j, with $Nj \leq i < N(j+1)$. So, if we have, let's say, 8 original charges $Q_1 \cdots Q_8$, and we decide to split each of these original charges into $N = 4$ sub-charges, then we get 32 sub-charges $q_1 \cdots q_{32}$, with sub-charges q_1, q_2, q_3, q_4 being in the same cell as initially Q_1, sub-charges q_5, q_6, q_7, q_8 being in the same cell as initially Q_2, and so on. These sub-charges are then evenly distributed along the x-axes.

(2) Each of the sub-charges get the initial value $q_i = \frac{Q_j}{N}$ for every $Nj \leq i < N(j+1)$. So, in our example the initial values are set to $q_1 = q_2 = q_3 = q_4 = \frac{Q_1}{4}, q_5 = q_6 = q_7 = q_8 = \frac{Q_2}{4}, \cdots, q_{29} = q_{30} = q_{31} = q_{32} = \frac{Q_8}{4}$ (see Figure 2).

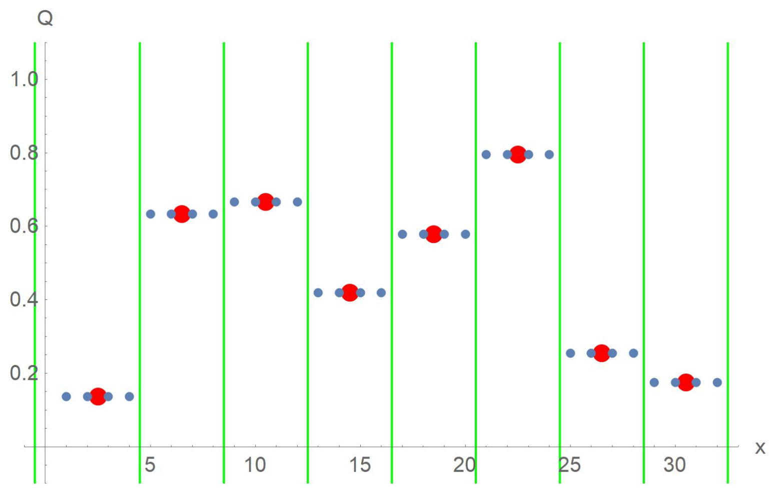

Figure 2: Initial distribution of the smaller sub-charges (blue dots). Please note that the sub-charges (blue dots) are plotted with four times their actual values to better set them in visual context with the original charges (red dots).

(3) This third step, if applied repeatedly, ensures that the discrete second derivative of the final distribution becomes constant. First, we randomly select a sub-charge q_i, excluding charges on the rightmost border of a cell from this selection process (i.e. excluding charges where $i + 1$ is a multiple of N). We now want to find a value Δq by which we modify $q_i \to q'_i$ and $q_{i+1} \to q'_{i+1}$ as follows:

$$q'_i = q_i - \Delta q \tag{1}$$

$$q'_{i+1} = q_{i+1} + \Delta q \tag{2}$$

This leaves the total charge in the cell unchanged.

After repeating this third step often enough, we want the discrete second derivative of the final distribution to be constant. Therefore, we expect that the gradient $\frac{q_{i+1}-q_i}{\Delta x}$ will eventually become the mean value of the left-side gradient $\frac{q_i-q_{i-1}}{\Delta x}$ and the right-side gradient $\frac{q_{i+2}-q_{i+1}}{\Delta x}$:

$$\frac{q_{i+1}^{final} - q_i^{final}}{\Delta x} = \frac{1}{2}\left(\frac{q_i^{final} - q_{i-1}^{final}}{\Delta x} + \frac{q_{i+2}^{final} - q_{i+1}^{final}}{\Delta x}\right) \tag{3}$$

As we don't know the final values for all q_i's yet, we have to use the following iterative equation in each iteration step instead:

$$\frac{q'_{i+1} - q'_i}{\Delta x} = \frac{1}{2}\left(\frac{q_i - q_{i-1}}{\Delta x} + \frac{q_{i+2} - q_{i+1}}{\Delta x}\right) \tag{4}$$

All sub-charges q_i are evenly distributed along the x-axes, therefore we can multiply equation (4) with Δx and get

$$q'_{i+1} - q'_i = \frac{1}{2}\left(q_i - q_{i-1} + q_{i+2} - q_{i+1}\right) \tag{5}$$

Inserting (1) and (2) on the left hand side of equation (5) leads to

$$q_{i+1} + \Delta q - q_i + \Delta q = \frac{1}{2}\left(q_i - q_{i-1} + q_{i+2} - q_{i+1}\right) \tag{6}$$

which can eventually be transformed into

$$\Delta q = \frac{q_{i+2} - 3q_{i+1} + 3q_i - q_{i-1}}{4} \tag{7}$$

We use this Δq to modify charges q_i and q_{i+1} as shown in equation (1) and (2).

This whole step (3) is to be repeated until equation (3) is fulfilled for all q_i within the required margin, except for the case when q_i is the rightmost charge in a cell, and also except for all q_i with $i = 1$, $i = N - 1$ or $i = N$, as we don't have charges q_0, q_{N+1} or q_{N+2} available to insert into formula (7).

(4) This last step, if applied repeatedly, ensures that the discrete *fourth* derivative of the final distribution becomes constant. As before, we start by randomly selecting a sub-charge q_i, excluding charges on the rightmost border of a cell from this selection process. And, again we want to find a value Δq by which to modify $q_i \to q'_i$ and $q_{i+1} \to q'_{i+1}$ as shown in equations (1) and (2), thereby leaving the total charge in a cell always unchanged.

After repeating this last step often enough, we want the discrete fourth derivative of the final distribution to be constant. Therefore, we expect that the third order finite difference $\delta_i^{(3)}$ will eventually become the mean value of the left-side third order finite difference $\delta_{i-1}^{(3)}$ and the right-side third order finite difference $\delta_{i+1}^{(3)}$:

$$\delta_i^{(3)final} = \frac{1}{2}\left(\delta_{i-1}^{(3final)} + \delta_{i+1}^{(3)final}\right) \tag{8}$$

The third order final difference is defined as

$$\delta_i^{(3)} = \sum_{k=0}^{3} (-1)^{n-k} \binom{n}{k} q_{i-1+k} \tag{9}$$

which can be resolved into

$$\delta_i^{(3)} = q_{i+2} - 3q_{i+1} + 3q_i - q_{i-1} \tag{10}$$

By inserting this into equation (8), we get

$$q_{i+2} - 3q_{i+1} + 3q_i - q_{i-1} = \frac{q_{i+3} - 3q_{i+2} + 3q_{i+1} - q_i + q_{i+1} - 3q_i + 3q_{i-1} - q_{i-2}}{2} \tag{11}$$

Again, this is the set of equations for the final solution. However, as we don't know the final values for all q_i's yet, we have to use the following iterative equation for any randomly chosen charge q_i and its neighbor q_{i+1} in each iteration step instead:

$$q_{i+2} - 3q'_{i+1} + 3q'_i - q_{i-1} = \frac{q_{i+3} - 3q_{i+2} + 3q_{i+1} - q_i + q_{i+1} - 3q_i + 3q_{i-1} - q_{i-2})}{2} \tag{12}$$

Inserting (1) and (2) on the left hand side of equation (12) leads to

$$\begin{aligned} q_{i+2} - 3\left(q_{i+1} + \Delta q\right) + 3\left(q_i - \Delta q\right) - q_{i-1} = \\ \frac{q_{i+3} - 3q_{i+2} + 3q_{i+1} - q_i + q_{i+1} - 3q_i + 3q_{i-1} - q_{i-2})}{2} \end{aligned} \tag{13}$$

which can eventually be transformed into

$$\Delta q = \frac{q_{i-2} - 5q_{i-1} + 10q_i - 10q_{i+1} + 5q_{i+2} - q_{i+3}}{12} \tag{14}$$

We use this Δq to modify charges q_i and q_{i+1} as shown in equation (1) and (2).

This whole step (4) is to be repeated until equation (11) is fulfilled for all q_i within the required margin, except for the case when q_i is the rightmost charge in a cell, and also except for all $i < 3$ or $i > N - 3$, as we don't have charges q_{-1}, q_0, q_{N+1}, q_{N+2} or q_{N+3} available to insert into formula (14).

2.1.2 Implementation in Python

Appendix 6.1 shows our Python 3.6 implementation of the algorithm explained in the previous chapter. All source code line references in this chapter refer to that listing.

We used our implementation to generate 2,000,000 records of training data (defined in line 8), each record representing 8 original (coarse) random charges (defined in line 10) as training input for a neural network. Each original charge is then split into 4 sub-charges (defined in line 11), so that the resulting output consists of 32 smoothly distributed charges representing a target configuration to which the neural network is to be trained.

`chrg_generator(n)`

Generator `chrg_generator(n)` (lines 15-40) returns n training data sets, each containing 8 original (coarse) random charges, and 32 corresponding refined charges. Refined charges are initialized to have the same value as the original charge in the same cell (line 24), which means that the division by 4 has been skipped in order to ensure that both original and refined have the same magnitude between 0 and 1 (with refined charges occasionally having values slightly below 0 or above 1).

Line 27-32: The first refinement step (ensuring that the discrete second derivative becomes constant) is performed 50 times (each time on a randomly chosen charge), before the maximum deviation is checked. This step is repeated until all charges are within the chosen absolute deviation of $2.5 \cdot 10^{-6}$ (as defined in line 13).

Line 34-39: The second refinement step (ensuring that the discrete fourth derivative becomes constant) is handled in the same way.

`deviationOK(q, step)`

Lines 41-51: Function `deviationOK(q, step)` receives an array `q` of refined charges, and returns `true`, if all charges are within the chosen deviation, i.e.: if all Δq by which the charges should be readjusted are smaller than `maxErr`. The second parameter `step` defines whether the deviation is to be checked with regards to the first or the second refinement step.

`refine(chrg, step, i=-1)`

Lines 53-68: Function `refine(chrg, step, i=-1)` receives an array `chrg` of to-be-refined charges. If parameter `i` is passed to the function, then the i-th and $(i+1)$-th charge is adjusted by Δq. If parameter `i` is not passed, a charge in array `chrg` will be randomly chosen. If the charge defined by `i`, or the randomly chosen charge, happens to be the rightmost charge in a cell, then there are no modifications. This is ensured, because function `dq(chrg, i, step)`, representing Δq, returns zero in such cases (and also in case where the charge in question happens to be a boundary charge not captured by the algorithm). Parameter `step` defines whether the refinement is to be done with respect to the first or the second refinement.

`dq_func(chrg, i, step)`

Lines 70-97: Function `dq_func(chrg, i, step)` corresponds to Δq in the refinement algorithm. In its first parameter, it receives an array `chrg` of to-be-refined charges. The second parameter `i` defines which charge in the array (together with the neighboring charge $i + 1$) is to be modified. The last parameter `step` defines, whether Δq is to be calculated with regards to the first or the second refinement step. If `i` defines a rightmost charge in a cell (line 82), or a boundary charge not captured by the algorithm (line 87 and line 93), then zero is returned.

`Main Program`

In lines 104-114, arrays `train_data` and `train_targets` are filled with random (coarse) charge distributions and corresponding refined charge distributions produced by generator `chrg_generator(n)`. Progress information is printed every 100 generated records.

In lines 116-122, the refined charges are re-scaled so that the re-scaled values never leave the range between 0 and 1, in order to match a typical value range for many neural networks's output layer.

In line 124-138, the generated data is stored into two files `train_data.pkl` and `train_targets.pkl`.

2.2 Exploration of Multiple Deep Learning Configurations

2.2.1 Implementation in Python

After having created the training data, we implemented the "Charge Refine Deep Learning Explorer", which is a piece of software for testing various Deep Learning models against this training set. The Python 3.6 source code, utilizing Keras 2.2.4, is listed in Appendix 6.2. All source code line references in this chapter refer to that listing.

`Network Parameter Definition`

In lines 14-20, some general network (meta-)parameters are defined, namely

- `trainSetSize=1500000` ... defines the number of records from the created training set used for actually training the network
- `trainSetSize=300000` ... defines the number of records from the created training set used for validating the network after each epoch
- `testSetSize=200000` ... defines the number of records from the created training set used for testing the accuracy of the fully trained network.
- `bSize=5000` ... in Keras, the batch size is the number of training examples in one forward/backward pass, i.e. the number of samples to work through before the internal model parameters are updated. The higher the batch size, the more memory space is needed.
- `myoptimizer='rmsprop'` ... this defines the optimizer by which the network's weights are updated during the training phase. The selected optimizer uses "Root Mean Square Propagation", which is very commonly used and combines the the concepts of exponential moving average of the past gradients and adapting learning rate.
- `maxepochs=2000` ... The number of epochs defines the number times that the learning algorithm will work through the entire training dataset. This parameter defines that training should stop after 2000 epochs at the latest.
- `mypat=200` ... This defines the number of epochs after which training will be stopped prematurely if there is no further improvement.

`Physics Parameter Definition`

In lines 22-23, the parameters of the physics problem are defined, namely:

- `numCells=8` ...This defines the number of Wigner-Seitz-cells), i.e. the number of (coarse) original charges in the training data set.
- `pointsPerCell=4` ...This defines the number of (refined) sub-charges each original (coarse) charge has been split into in the training data set.

`createModel(nh1, nh2, nh3, nactfunc, actfuncin=True, actfuncout=False)`

Lines 31-77: This function creates and returns a dense deep learning model with up to three hidden layers. The size of the input layer is unchangeable and defined by the number of original charges `numCells=8`. Also, the size of the output layer is fixed at `numCells*pointsPerCell` (which is 32, in our case). However, the number and size of hidden layers is flexible and can be defined by passing parameters `nh1`, `nh2`, `nh3` (defining the number of neurons in hidden layer 1, 2 and 3).

Parameter `nactfunc` may take on values between `0` and `10` and defines the network's activation function (line 35-56). Parameter `actfuncin` is boolean (default: `True`), and defines whether the chosen activation function is also applied to the input layer (if it is `False`, then the input layer always has just a linear activation function). Boolean parameter `actfuncout` (default: `False`) does the same with the output layer.

`plotToFile(history, round, step)`

Lines 80-105: This function expects a `History` object as returned by Keras' `mode.fit` method, and plots a graph of the training progress (training and validation loss) into a file. The integer parameter `round` just influences the file name (line 102), and reflects what number model has been tested. Finally, the integer parameter `step` may take on values between `1` and `4`. If it is `1`, then the whole curve is plotted. If it is `2`, `3` or `4`, then the first 10, 40, or 150 data points are skipped in the graphic.

`MAIN PROGRAM`

The main program starts at line 107. In lines 115-128, the complete training data set is loaded from the file. In line 130-140, the data set is split between actual training data, validation data and test data.

In lines 142-160, a log file is created, containing a header with general parameter information, and a headline for the information in the lines to come.

In lines 162-176, the number and type of the models to be tested are defined. The parameters are the same as in `createModel(...)`, namely `nh1`, `nh2`, `nh3` as the number of neurons in hidden layers 1, 2 and 3 (zero for no layer), and `nactfunc`, `actfuncin`, and `actfuncout` to define which activation function to be used on what layers.

In lines 178-228, the actual simulations are executed, with the main loop starting in line 191. One by one, each model with parameters as defined before, is generated (line 195), compiled (lines 196-197) and trained (lines 199-202). After that, the model is scored against the test data set (line 205), and information is printed on screen and written to the log file (lines 206-228). Eventually, in lines 230-236, graphs of the training progress are plotted into files.

2.2.2 Results

In the first experiment, we ran multiple simulations with 1, 2 and 3 hidden layers, where the number of neurons in these layers were permutations of 4, 8, 16, 32, 64 and 128. In all these simulations we used the `sigmoid` activation function.

It turned out that all these networks could be trained to learn the training set, with mean absolute errors between 0.12 and 0.08 in less than 90 epochs (the `patience` parameter was set to 6 as to allow the simulations to end quickly for a fast screening of the various architectures). As expected, networks having a layer with just 4 neurons performed generally below average. Also, networks with one or two hidden layers performed better on average than networks with three hidden layers. It was also interesting that we never observed any overfitting.

In follow-up experiments we continued testing with network architectures that have proven to be above average in the first run. We increased the number of epochs (by setting the `patience` parameter gradually to higher values), and we also varied the activation function. It turned out that from all *non-linear* activation functions, the `tanh`-function performed best, followed by `sigmoid`, `hard_sigmoid`, and `elu`. With all other nonlinear activation functions the training results were non-satisfactory with the given architectures. Also, networks where input- and output layers did not have an activation function (i.e. had a linear activation function) performed better on average.

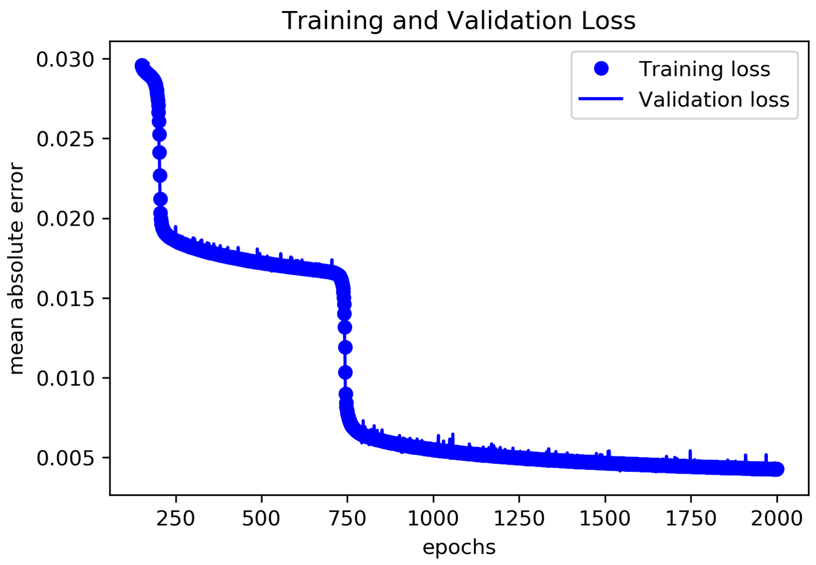

Figure 3: Training and validation loss (mean absolute error) development while training a dense deep network with 8-16-32-16-32 neurons and `sigmoid` activation function over 2000 epochs.

In these experiments, a `tanh`-network with two hidden layers having 128 and 16 neurons, trained over 101 epochs, achieved the best mean average error of 0.004. However, a much simpler `tanh`-network with just *one* hidden layer with 8 neurons performed almost equally well with a mean average error 0.0052.

Although they are supposedly a core ingredient of deep learning architectures (see e.g. [Goodfellow et al, 2016, p. 167])), we eventually decided to train networks where we have abandoned all non-linear activation functions in favor of the linear activation function. We also allowed for a network without any hidden layer. From all these networks, the most simple network, consisting of just 8 input and 32 output neurons (no hidden layers) performed best after being trained over 2000 epochs, reaching a mean average error on the test data of just 0.0011 (see Figure 4).

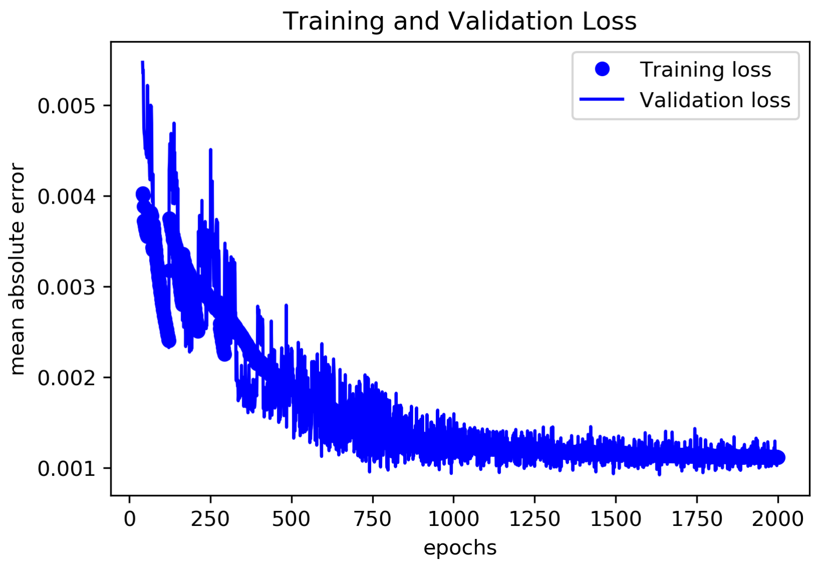

Figure 4: Training and validation loss (mean absolute error) development while training a dense network with 8 input and 32 output neurons (no hidden layers) and `linear` activation function over 2000 epochs.

3 The Linear Algebra Approach

3.1 From Deep Learning to Linear Algebra

Obviously, because of the linear activation function, the network from Figure (4), having just an 8-neuron input layer and a densely connected 32-neuron output layer, can be expressed in a simple vector/matrix equation.

Let $\vec{Q}$ be the 8×1 input vector of (coarse) original charges. The first layer of the trained network can be represented by a 8×8 weight matrix $\underline{\underline{W_1}}$ and a 8×1 bias vector $\vec{b}_1$, so that the 8×1 output vector $\vec{o}_1$ of the first layer can be written as

$$\vec{o}_1 = \underline{\underline{W_1}}\vec{Q} + \vec{b}_1 \tag{15}$$

Let $\vec{q}$ be the to-be-calculated 32×1 vector of refined small charges. We want this vector to be the output of the second layer. Like the first layer, the second layer of the trained network can be represented by a 32×8 weight matrix $\underline{\underline{W_2}}$ and a 32×1 bias vector $\vec{b}_2$. As the output vector $\vec{o}_1$ of the first layer is the input vector of the second layer, we can write

$$\vec{q} = \underline{\underline{W_2}}\vec{o}_1 + \vec{b}_2 \tag{16}$$

By substituting $\vec{o}_1$ with the result from equation (15), we finally get

$$\vec{q} = \underline{\underline{W_2}}\left(\underline{\underline{W_1}}\vec{Q} + \vec{b}_1\right) + \vec{b}_2 \tag{17}$$

Considering that the split charges in the training dataset were not divided by 4, and further considering that the training dataset is re-sized (see source code lines 115-121 on page 69), the correct final formula is

$$\vec{q} = \frac{1}{4}\left(2\left(\underline{\underline{W_2}}\left(\underline{\underline{W_1}}\vec{Q} + \vec{b}_1\right) + \vec{b}_2\right) - \begin{pmatrix} 0.5 \\ \vdots \\ 0.5 \end{pmatrix}\right) \tag{18}$$

These are the actual matrices and vectors as retrieved from a trained network (rounded to five digits after the decimal point):

$$\underline{\underline{W_1}} = \begin{pmatrix}
-0.23201 & -0.09500 & -0.18980 & -0.20857 & -0.13743 & -0.24721 & -0.24588 & -0.11497 \\
0.83026 & 0.49651 & -0.41574 & -0.79812 & 0.18860 & -0.10128 & 0.24538 & -0.51374 \\
0.81205 & -0.35277 & -0.01770 & 0.67868 & 0.02250 & -0.03245 & -0.89922 & -0.43290 \\
0.09440 & -0.78898 & -0.50042 & 0.31114 & 0.57818 & -0.56365 & 0.67051 & -0.07993 \\
0.11048 & -0.67576 & -0.00760 & -0.25418 & -0.47597 & 0.71256 & 0.18709 & -0.23155 \\
-0.63754 & 0.13760 & 0.17649 & 0.03006 & 0.78638 & 0.45352 & -0.10451 & -0.99403 \\
0.08440 & -0.10035 & 0.95792 & 0.00775 & -0.44585 & -0.61243 & 0.42449 & -0.57333 \\
0.17292 & -0.50736 & 0.67976 & -0.76484 & 0.68332 & -0.06788 & -0.45818 & 0.48310
\end{pmatrix}$$

$$\underline{\underline{W_2}} = \begin{pmatrix}
-0.37280 & 0.20164 & 0.17900 & 0.01525 & 0.02771 & -0.13803 & 0.01954 & 0.04117 \\
-0.39945 & 0.20787 & 0.21252 & 0.04556 & 0.07320 & -0.15716 & 0.03436 & 0.07578 \\
-0.38700 & 0.20508 & 0.19449 & 0.03114 & 0.05230 & -0.14704 & 0.02970 & 0.06403 \\
-0.33200 & 0.19186 & 0.12743 & -0.03208 & -0.04342 & -0.10714 & -0.00616 & -0.01223 \\
-0.25288 & 0.16871 & 0.02983 & -0.12346 & -0.18045 & -0.04828 & -0.05468 & -0.12227 \\
-0.19000 & 0.13716 & -0.06000 & -0.20229 & -0.28451 & 0.01296 & -0.06831 & -0.17996 \\
-0.16642 & 0.09801 & -0.11750 & -0.24235 & -0.31373 & 0.05563 & -0.02421 & -0.14796 \\
-0.18825 & 0.05178 & -0.13084 & -0.23736 & -0.25991 & 0.07827 & 0.07031 & -0.03150 \\
-0.24450 & -0.00152 & -0.10079 & -0.19346 & -0.14884 & 0.07750 & 0.18644 & 0.11941 \\
-0.30716 & -0.06332 & -0.04195 & -0.13948 & -0.03869 & 0.05886 & 0.26547 & 0.21186 \\
-0.35718 & -0.12917 & 0.03078 & -0.09094 & 0.03198 & 0.02960 & 0.27786 & 0.19741 \\
-0.38375 & -0.19150 & 0.10322 & -0.05402 & 0.04701 & -0.00189 & 0.21986 & 0.08051 \\
-0.38359 & -0.23216 & 0.15648 & -0.02019 & 0.00501 & -0.02153 & 0.11643 & -0.08960 \\
-0.36382 & -0.23266 & 0.17641 & 0.02562 & -0.07015 & -0.01709 & 0.01529 & -0.21039 \\
-0.33096 & -0.18771 & 0.15776 & 0.08793 & -0.15578 & 0.01560 & -0.05169 & -0.22978 \\
-0.29522 & -0.10675 & 0.10781 & 0.15693 & -0.22899 & 0.07162 & -0.07780 & -0.14076 \\
-0.26624 & -0.01268 & 0.04364 & 0.20845 & -0.26644 & 0.13593 & -0.07638 & 0.01891 \\
-0.25416 & 0.05250 & -0.00209 & 0.20545 & -0.24131 & 0.18271 & -0.08259 & 0.16049 \\
-0.26392 & 0.07117 & -0.01775 & 0.13709 & -0.15036 & 0.20022 & -0.11219 & 0.23114 \\
-0.29488 & 0.04434 & -0.00560 & 0.01877 & -0.00947 & 0.18593 & -0.16472 & 0.21775 \\
-0.34080 & -0.00661 & 0.01843 & -0.10938 & 0.14731 & 0.14910 & -0.21340 & 0.12960 \\
-0.38976 & -0.04036 & 0.01966 & -0.17844 & 0.26496 & 0.11097 & -0.21785 & 0.02027 \\
-0.43159 & -0.03668 & -0.01548 & -0.15817 & 0.31318 & 0.08145 & -0.16128 & -0.07587 \\
-0.45845 & 0.00338 & -0.08588 & -0.05457 & 0.28565 & 0.06325 & -0.05388 & -0.13660 \\
-0.46171 & 0.05988 & -0.16880 & 0.08889 & 0.20170 & 0.04714 & 0.06781 & -0.15345 \\
-0.43672 & 0.08876 & -0.22419 & 0.19112 & 0.10253 & 0.01579 & 0.13800 & -0.13051 \\
-0.38573 & 0.06980 & -0.23377 & 0.20978 & 0.01723 & -0.03767 & 0.12797 & -0.07540 \\
-0.31756 & 0.00571 & -0.19674 & 0.14598 & -0.04022 & -0.10870 & 0.04075 & -0.00368 \\
-0.24792 & -0.07773 & -0.13403 & 0.03791 & -0.07191 & -0.18034 & -0.08090 & 0.06675 \\
-0.20141 & -0.13685 & -0.08937 & -0.03926 & -0.08809 & -0.22960 & -0.16495 & 0.11670 \\
-0.18958 & -0.14830 & -0.08119 & -0.05286 & -0.09633 & -0.24031 & -0.18200 & 0.13125 \\
-0.21283 & -0.12134 & -0.10137 & -0.01810 & -0.08507 & -0.21727 & -0.14212 & 0.10492
\end{pmatrix}$$

$$\vec{b}_1 = \begin{pmatrix} 0.33668 \\ 0.03266 \\ 0.11375 \\ 0.14340 \\ 0.32480 \\ 0.07632 \\ 0.13475 \\ -0.11565 \end{pmatrix} \quad \vec{b}_2 = \begin{pmatrix} 0.34976 \\ 0.33894 \\ 0.34399 \\ 0.36639 \\ 0.39867 \\ 0.42439 \\ 0.43396 \\ 0.42489 \\ 0.40172 \\ 0.37605 \\ 0.35625 \\ 0.34669 \\ 0.34708 \\ 0.35531 \\ 0.36727 \\ 0.38109 \\ 0.39280 \\ 0.39767 \\ 0.39366 \\ 0.38103 \\ 0.36234 \\ 0.34247 \\ 0.32567 \\ 0.31617 \\ 0.31504 \\ 0.32510 \\ 0.34581 \\ 0.37331 \\ 0.40058 \\ 0.41935 \\ 0.42426 \\ 0.41460 \end{pmatrix}$$

Please note that the matrices and vectors printed above are not a unique, reproducible solution. Whenever re-trained, the network ends up with significantly different weight-matrices and bias vectors. This is a clear indication for the structure of this solution to be still redundant, and the exact linear solution to be even simpler.

3.2 The Exact Linear Solution

3.2.1 Exact Solution for Constant Second Derivative

Let us consider a very simple toy-problem where 3 coarse original charges Q_1, Q_2, Q_3 are to be split into 4 sub-charges each, thereby producing charges $q_1 \cdots q_{12}$, so that the discrete second derivative of the final charge distribution $q_1 \cdots q_{12}$ becomes constant within each Wigner-Seitz cell (see Fig. 5).

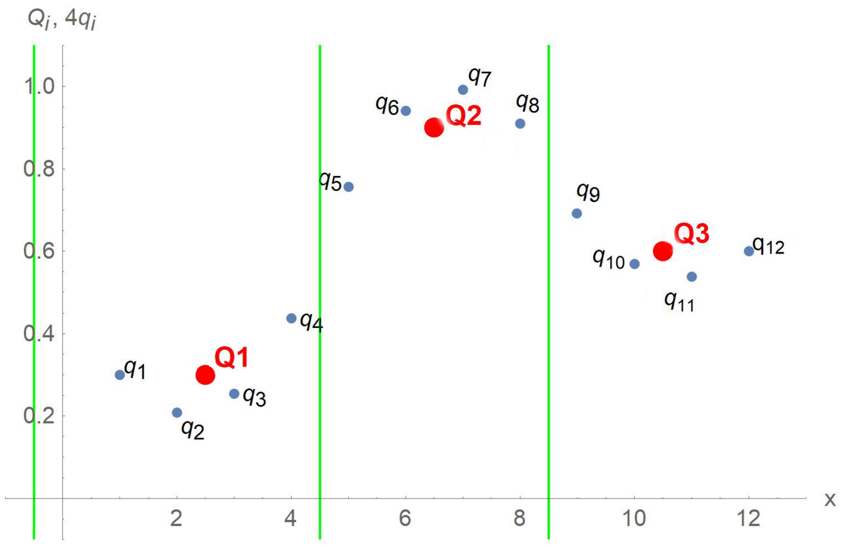

Figure 5: An exemplary toy-problem: 3 original charges Q_1, Q_2, Q_3 (red dots) are to be split into 4 sub-charges each, so that the discrete second derivative of the final charge distribution $q_1 \cdots q_{12}$ (blue dots) becomes constant within each Wigner-Seitz cell (green separation lines).

Considering formula (3), and further considering that under the algorithm explained in chapter 2.1.1 it is not allowed to simultaneously modify the rightmost charge q_i of a cell and the leftmost charge q_{i+1} of the neighboring cell, we can already write down the following (yet under-defined) system of equations:

$$\begin{aligned}
q_3 - q_2 &= \frac{1}{2}(q_4 - q_3 + q_2 - q_1) \\
q_4 - q_3 &= \frac{1}{2}(q_5 - q_4 + q_3 - q_2) \\
q_6 - q_5 &= \frac{1}{2}(q_7 - q_6 + q_5 - q_4) \\
q_7 - q_6 &= \frac{1}{2}(q_8 - q_7 + q_6 - q_5) \\
q_8 - q_7 &= \frac{1}{2}(q_9 - q_8 + q_7 - q_6) \\
q_{10} - q_9 &= \frac{1}{2}(q_{11} - q_{10} + q_9 - q_8) \\
q_{11} - q_{10} &= \frac{1}{2}(q_{12} - q_{11} + q_{10} - q_9)
\end{aligned} \tag{19}$$

These equations can be re-written as

$$
\begin{aligned}
\frac{1}{2}q_1 - \frac{3}{2}q_2 + \frac{3}{2}q_3 - \frac{1}{2}q_4 &= 0 \\
\frac{1}{2}q_2 - \frac{3}{2}q_3 + \frac{3}{2}q_4 - \frac{1}{2}q_5 &= 0 \\
\frac{1}{2}q_4 - \frac{3}{2}q_5 + \frac{3}{2}q_6 - \frac{1}{2}q_7 &= 0 \\
\frac{1}{2}q_5 - \frac{3}{2}q_6 + \frac{3}{2}q_7 - \frac{1}{2}q_8 &= 0 \\
\frac{1}{2}q_6 - \frac{3}{2}q_7 + \frac{3}{2}q_8 - \frac{1}{2}q_9 &= 0 \\
\frac{1}{2}q_8 - \frac{3}{2}q_9 + \frac{3}{2}q_{10} - \frac{1}{2}q_{11} &= 0 \\
\frac{1}{2}q_9 - \frac{3}{2}q_{10} + \frac{3}{2}q_{11} - \frac{1}{2}q_{12} &= 0
\end{aligned}
\tag{20}
$$

Throwing in the condition that, within each Wigner-Seitz cell, the sum of small charges must equal the original charge in this cell, we get three more equations:

$$
\begin{aligned}
q_1 + q_2 + q_3 + q_4 &= Q_1 \\
q_5 + q_6 + q_7 + q_8 &= Q_2 \\
q_9 + q_{10} + q_{11} + q_{12} &= Q_3
\end{aligned}
\tag{21}
$$

To make the system of equations complete, we need two more equations. These are provided by the initial value of the leftmost and rightmost charge (see step (2) of the algorithm in chapter 2.1.1):

$$
\begin{aligned}
q_1 &= \frac{1}{4}Q_1 \\
q_{12} &= \frac{1}{4}Q_3
\end{aligned}
\tag{22}
$$

The complete system of linear equations (20), (21) and (22) can be represented in the following single matrix/vector equation:

$$
\begin{pmatrix} \color{red}{Q_1} \\ \color{blue}{\frac{Q_1}{4}} \\ 0 \\ 0 \\ \color{red}{Q_2} \\ 0 \\ 0 \\ 0 \\ \color{red}{Q_3} \\ 0 \\ 0 \\ \color{blue}{\frac{Q_3}{4}} \end{pmatrix}
=
\begin{bmatrix}
\color{red}{1} & \color{red}{1} & \color{red}{1} & \color{red}{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\color{blue}{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & -\frac{3}{2} & \frac{3}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & -\frac{3}{2} & \frac{3}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \color{red}{1} & \color{red}{1} & \color{red}{1} & \color{red}{1} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{2} & -\frac{3}{2} & \frac{3}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{3}{2} & \frac{3}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{3}{2} & \frac{3}{2} & -\frac{1}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \color{red}{1} & \color{red}{1} & \color{red}{1} & \color{red}{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{3}{2} & \frac{3}{2} & -\frac{1}{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{3}{2} & \frac{3}{2} & -\frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \color{blue}{1}
\end{bmatrix}
\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \end{pmatrix}
\tag{23}
$$

In (23), all entries corresponding to the three equations in (21) are printed in red, and all entries corresponding to the two equations in (22) are printed in blue. All remaining entries in black color reflect the eight equations from (20).

If we call the matrix in equation (23) matrix $\underline{\underline{A}}$, then the solution we are looking for can be written as

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \end{pmatrix} = \underline{\underline{A}}^{-1} \begin{pmatrix} Q_1 \\ \frac{Q_1}{4} \\ 0 \\ 0 \\ Q_2 \\ 0 \\ 0 \\ 0 \\ Q_3 \\ 0 \\ 0 \\ \frac{Q_3}{4} \end{pmatrix} \tag{24}$$

Below is the inverse matrix $\underline{\underline{A}}^{-1}$, as explicitly calculated for our example:

$$\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1047}{3854} & \frac{91}{1927} & -\frac{1229}{1927} & -\frac{880}{1927} & -\frac{2}{47} & \frac{364}{1927} & \frac{8}{41} & \frac{200}{1927} & \frac{30}{1927} & -\frac{60}{1927} & -\frac{40}{1927} & -\frac{50}{1927} \\
\frac{1487}{3854} & -\frac{918}{1927} & \frac{349}{1927} & -\frac{440}{1927} & -\frac{1}{47} & \frac{182}{1927} & \frac{4}{41} & \frac{100}{1927} & \frac{15}{1927} & -\frac{30}{1927} & -\frac{20}{1927} & -\frac{25}{1927} \\
\frac{660}{1927} & -\frac{1100}{1927} & \frac{880}{1927} & \frac{1320}{1927} & \frac{3}{47} & -\frac{546}{1927} & -\frac{12}{41} & -\frac{300}{1927} & -\frac{45}{1927} & \frac{90}{1927} & \frac{60}{1927} & \frac{75}{1927} \\
\frac{273}{1927} & -\frac{455}{1927} & \frac{364}{1927} & \frac{546}{1927} & \frac{10}{47} & -\frac{1820}{1927} & -\frac{40}{41} & -\frac{1000}{1927} & -\frac{150}{1927} & \frac{300}{1927} & \frac{200}{1927} & \frac{250}{1927} \\
\frac{9}{1927} & -\frac{15}{1927} & \frac{12}{1927} & \frac{18}{1927} & \frac{27}{94} & -\frac{60}{1927} & -\frac{27}{41} & -\frac{880}{1927} & -\frac{132}{1927} & \frac{264}{1927} & \frac{176}{1927} & \frac{220}{1927} \\
-\frac{132}{1927} & \frac{220}{1927} & -\frac{176}{1927} & -\frac{264}{1927} & \frac{27}{94} & \frac{880}{1927} & \frac{27}{41} & \frac{60}{1927} & \frac{9}{1927} & -\frac{18}{1927} & -\frac{12}{1927} & -\frac{15}{1927} \\
-\frac{150}{1927} & \frac{250}{1927} & -\frac{200}{1927} & -\frac{300}{1927} & \frac{10}{47} & \frac{1000}{1927} & \frac{40}{41} & \frac{1820}{1927} & \frac{273}{1927} & -\frac{546}{1927} & -\frac{364}{1927} & -\frac{455}{1927} \\
-\frac{45}{1927} & \frac{75}{1927} & -\frac{60}{1927} & -\frac{90}{1927} & \frac{3}{47} & \frac{300}{1927} & \frac{12}{41} & \frac{546}{1927} & \frac{660}{1927} & -\frac{1320}{1927} & -\frac{880}{1927} & -\frac{1100}{1927} \\
\frac{15}{1927} & -\frac{25}{1927} & \frac{20}{1927} & \frac{30}{1927} & -\frac{1}{47} & -\frac{100}{1927} & -\frac{4}{41} & -\frac{182}{1927} & \frac{1487}{3854} & \frac{440}{1927} & -\frac{349}{1927} & -\frac{918}{1927} \\
\frac{30}{1927} & -\frac{50}{1927} & \frac{40}{1927} & \frac{60}{1927} & -\frac{2}{47} & -\frac{200}{1927} & -\frac{8}{41} & -\frac{364}{1927} & \frac{1047}{3854} & \frac{880}{1927} & \frac{1229}{1927} & \frac{91}{1927} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \tag{25}$$

However, it would be much more elegant, and also more compact, to write the solution in the form of a 12×3 matrix $\underline{\underline{M_1}}$ so that

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \end{pmatrix} = \underline{\underline{M_1}} \begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \end{pmatrix} \tag{26}$$

Matrix $\underline{\underline{M_1}}$ can easily be derived from solution (24) and (25). When setting $Q_1 = 1$, $Q_2 = 0$, and $Q_3 = 0$, (24) returns a vector representing the first column of matrix $\underline{\underline{M_1}}$. Similarly, setting $Q_1 = 0$, $Q_2 = 1$, and $Q_3 = 0$ produces the second column of matrix $\underline{\underline{M_1}}$; and when we set $Q_1 = 0$, $Q_2 = 0$, and $Q_3 = 1$ we get the third column of matrix $\underline{\underline{M_1}}$.

The final and most compact solution with regards to our special-case problem can therefore be written as:

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \end{pmatrix} = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ \frac{2185}{7708} & -\frac{2}{47} & \frac{35}{3854} \\ \frac{514}{1927} & -\frac{1}{47} & \frac{35}{7708} \\ \frac{385}{1927} & \frac{3}{47} & -\frac{105}{7708} \\ \frac{637}{7708} & \frac{10}{47} & -\frac{175}{3854} \\ \frac{21}{7708} & \frac{27}{94} & -\frac{77}{1927} \\ -\frac{77}{1927} & \frac{27}{94} & \frac{21}{7708} \\ -\frac{175}{3854} & \frac{10}{47} & \frac{637}{7708} \\ -\frac{105}{7708} & \frac{3}{47} & \frac{385}{1927} \\ \frac{35}{7708} & -\frac{1}{47} & \frac{514}{1927} \\ \frac{35}{3854} & -\frac{2}{47} & \frac{2185}{7708} \\ 0 & 0 & \frac{1}{4} \end{bmatrix} \begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \end{pmatrix} \tag{27}$$

Based on this descriptive toy-example, we can now easily deduct the algorithm for a general $\underline{\underline{M_1}}$ matrix of arbitrary size:

(1) Let Q_j be the single original charge in the j-th Wigner-Seitz cell, and j_{max} the total number of cells. Further, let n be the number of (smaller) sub-charges to be created for each original charge Q_j. Then $m = nj_{max}$ is the total number of smaller sub-charges $q_1 \dots q_m$. In the first step, we generate a Matrix $\underline{\underline{A}}$ of size $m \times m$ with elements $a_{i,j}$, and we set all matrix elements $a_{i,j} = 0$.

(2) In each $(kn+1)$-th line of matrix $\underline{\underline{A}}$ (with $k \in \mathbb{N}_{\geq 0}$), we set $a_{kn+1,kn+1} = 1$, $a_{kn+1,kn+2} = 1$, $a_{kn+1,kn+3} = 1$, and $a_{kn+1,kn+4} = 1$ (this corresponds to the red entries in equation (23)).

(3) We then set the first entry in the second line $a_{2,1} = 1$ and the bottom-right entry $a_{m,m} = 1$ (this corresponds to the blue entries in equation (23)).

(4) In all lines i hitherto not yet modified, we set $a_{i,i-2} = \frac{1}{2}$, $a_{i,i-1} = -\frac{3}{2}$, $a_{i,i} = \frac{3}{2}$, and $a_{i,i+1} = -\frac{1}{2}$ (this corresponds to the black entries in equation (23)).

(5) We calculate $\underline{\underline{A}}^{-1}$ by inverting matrix $\underline{\underline{A}}$.

(6) We generate a row-vector $\vec{Q}$ with m lines, and we set all entries to zero, except for every $(kn+1)$-th entry (with $k \in \mathbb{N}_{\geq 0}$), which we define to be Q_{k+1}. Additionally, we define the second entry to be $\frac{Q_1}{n}$, and the last entry to be $\frac{Q_{j_{max}}}{n}$.

(7) Let $\vec{m}_j$ be the j-th column of matrix $\underline{\underline{M_1}}$. We create Matrix $\underline{\underline{M_1}}$ by calculating all columns $\vec{m}_j$ according to the following rule: $\vec{m}_j = \underline{\underline{A}}^{-1}\vec{Q}|_{Q_i=1 \; if \; i=j \; else \; 0}$

3.2.2 Exact Solution for Constant Fourth Derivative

Let us stick with our exemplary toy-problem: Let there be still 3 coarse original charges Q_1, Q_2, Q_3 to be split into 4 sub-charges each. But now, we want the discrete *fourth* derivative of the final charge distribution $q_1 \cdots q_{12}$ to become constant within each Wigner-Seitz cell (see Fig. 5).

Considering formula (11), and further considering that under the algorithm explained in chapter 2.1.1 it is not allowed to simultaneously modify the rightmost charge q_i of a cell and the leftmost charge q_{i+1} of the neighboring cell, we can again write down a yet under-defined system of equations:

$$\begin{aligned}
q_5 - 3q_4 + 3q_3 - q_2 &= \frac{1}{2}(q_6 - 3q_5 + 3q_4 - q_3 + q_4 - 3q_3 + 3q_2 - q_1) \\
q_7 - 3q_6 + 3q_5 - q_4 &= \frac{1}{2}(q_8 - 3q_7 + 3q_6 - q_5 + q_6 - 3q_5 + 3q_4 - q_3) \\
q_8 - 3q_7 + 3q_6 - q_5 &= \frac{1}{2}(q_9 - 3q_8 + 3q_7 - q_6 + q_7 - 3q_6 + 3q_5 - q_4) \\
q_9 - 3q_8 + 3q_7 - q_6 &= \frac{1}{2}(q_{10} - 3q_9 + 3q_8 - q_7 + q_9 - 3q_8 + 3q_7 - q_6) \\
q_{11} - 3q_{10} + 3q_9 - q_8 &= \frac{1}{2}(q_{12} - 3q_{11} + 3q_{10} - q_9 + q_{11} - 3q_{10} + 3q_9 - q_8)
\end{aligned} \tag{28}$$

This can be re-written as

$$\begin{aligned}
\frac{1}{2}q_1 - \frac{5}{2}q_2 + 5q_3 - 5q_4 + \frac{5}{2}q_5 - \frac{1}{2}q_6 &= 0 \\
\frac{1}{2}q_3 - \frac{5}{2}q_4 + 5q_5 - 5q_6 + \frac{5}{2}q_7 - \frac{1}{2}q_8 &= 0 \\
\frac{1}{2}q_4 - \frac{5}{2}q_5 + 5q_6 - 5q_7 + \frac{5}{2}q_8 - \frac{1}{2}q_9 &= 0 \\
\frac{1}{2}q_5 - \frac{5}{2}q_6 + 5q_7 - 5q_8 + \frac{5}{2}q_9 - \frac{1}{2}q_{10} &= 0 \\
\frac{1}{2}q_7 - \frac{5}{2}q_8 + 5q_9 - 5q_{10} + \frac{5}{2}q_{11} - \frac{1}{2}q_{12} &= 0
\end{aligned} \tag{29}$$

Again, within each Wigner-Seitz cell, the sum of small charges must equal the original charge in this cell. We therefore can add three more equations to the system:

$$\begin{aligned}
q_1 + q_2 + q_3 + q_4 &= Q_1 \\
q_5 + q_6 + q_7 + q_8 &= Q_2 \\
q_9 + q_{10} + q_{11} + q_{12} &= Q_3
\end{aligned} \tag{30}$$

The value of the leftmost and rightmost charge will not be changed by the algorithm described in chapter 2.1.1, and therefore remains at the initial value of a quarter of the respective original charge, which gives us two more equations:

$$\begin{aligned}
q_1 &= \frac{1}{4}Q_1 \\
q_{12} &= \frac{1}{4}Q_3
\end{aligned} \tag{31}$$

We are still two equations short of getting a solvable system of equations. This is because not only the leftmost charge q_1 and the rightmost charge q_{12}, but also the second-from-left charge q_2, and second-from-right charge q_{11} cannot be modified by the constant-fourth-derivative-algorithm.

However, the original algorithm as described in chapter 2.1.1 first aligns all sub-charges q_i so that the *second* derivative becomes constant, before it continues to align the charges with respect to the *fourth* derivative. We want the method we are just developing to produce the same result as this algorithm. Therefore, the second-from-left charge q_2, and second-from-right charge q_{11} should be aligned in accordance with matrix $\underline{\underline{M_1}}$. Hence, we can read the missing two equations directly from the second and eleventh line of matrix $\underline{\underline{M_1}}$ in equation (27):

$$\begin{aligned} q_2 &= \frac{2185}{7708}Q_1 - \frac{2}{47}Q_2 + \frac{35}{3854}Q_3 \\ q_{11} &= \frac{35}{3854}Q_1 - \frac{2}{47}Q_2 + \frac{2185}{7708}Q_3 \end{aligned} \tag{32}$$

The now complete system of linear equations (29), (30), (31), and (32) can be represented by the following single matrix/vector equation:

$$\begin{pmatrix} Q_1 \\ \frac{Q_1}{4} \\ \frac{2185}{7708}Q_1 - \frac{2}{47}Q_2 + \frac{35}{3854}Q_3 \\ 0 \\ Q_2 \\ 0 \\ 0 \\ 0 \\ Q_3 \\ 0 \\ \frac{35}{3854}Q_1 - \frac{2}{47}Q_2 + \frac{2185}{7708}Q_3 \\ \frac{Q_3}{4} \end{pmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{5}{2} & 5 & -5 & \frac{5}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{5}{2} & 5 & -5 & \frac{5}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & -\frac{5}{2} & 5 & -5 & \frac{5}{2} & -\frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{5}{2} & 5 & -5 & \frac{5}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{5}{2} & 5 & -5 & \frac{5}{2} & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \end{pmatrix} \tag{33}$$

In (33), all entries corresponding to the three equations in (30) are printed in red, all entries corresponding to the two equations in (31) are printed in blue, and all entries corresponding to the two equations in (32) are printed in brown. All remaining entries in black color reflect the five equations from (29).

If we call the matrix in equation (33) matrix $\underline{\underline{B}}$, then then the solution we are looking for can be written as

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \end{pmatrix} = \underline{\underline{B}}^{-1} \begin{pmatrix} Q_1 \\ \frac{Q_1}{4} \\ \frac{2185}{7708}Q_1 - \frac{2}{47}Q_2 + \frac{35}{3854}Q_3 \\ 0 \\ Q_2 \\ 0 \\ 0 \\ 0 \\ Q_3 \\ 0 \\ \frac{35}{3854}Q_1 - \frac{2}{47}Q_2 + \frac{2185}{7708}Q_3 \\ \frac{Q_3}{4} \end{pmatrix} \tag{34}$$

This is the inverse matrix $\underline{\underline{B}}^{-1}$, explicitly calculated and written out for our example:

$$\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{12557}{30134} & -\frac{14489}{30134} & -\frac{2897}{30134} & \frac{1932}{15067} & -\frac{2}{61} & -\frac{1743}{15067} & -\frac{32}{247} & -\frac{1002}{15067} & \frac{514}{15067} & \frac{203}{15067} & -\frac{2043}{30134} & -\frac{825}{30134} \\
\frac{17577}{30134} & -\frac{15645}{30134} & -\frac{27237}{30134} & -\frac{1932}{15067} & \frac{2}{61} & \frac{1743}{15067} & \frac{32}{247} & \frac{1002}{15067} & -\frac{514}{15067} & -\frac{203}{15067} & \frac{2043}{30134} & \frac{825}{30134} \\
\frac{20251}{60268} & -\frac{16765}{60268} & -\frac{37681}{60268} & -\frac{1743}{15067} & \frac{47}{244} & \frac{14609}{30134} & \frac{147}{247} & \frac{9669}{30134} & -\frac{10371}{60268} & -\frac{1002}{15067} & \frac{20391}{60268} & \frac{8367}{60268} \\
\frac{45}{3172} & -\frac{23}{3172} & -\frac{155}{3172} & -\frac{11}{793} & \frac{75}{244} & \frac{175}{1586} & \frac{5}{13} & \frac{435}{1586} & -\frac{565}{3172} & -\frac{50}{793} & \frac{1065}{3172} & \frac{465}{3172} \\
-\frac{565}{3172} & \frac{465}{3172} & \frac{1065}{3172} & \frac{50}{793} & \frac{75}{244} & -\frac{435}{1586} & -\frac{5}{13} & -\frac{175}{1586} & \frac{45}{3172} & \frac{11}{793} & -\frac{155}{3172} & -\frac{23}{3172} \\
-\frac{10371}{60268} & \frac{8367}{60268} & \frac{20391}{60268} & \frac{1002}{15067} & \frac{47}{244} & -\frac{9669}{30134} & -\frac{147}{247} & -\frac{14609}{30134} & \frac{20251}{60268} & \frac{1743}{15067} & -\frac{37681}{60268} & -\frac{16765}{60268} \\
-\frac{514}{15067} & \frac{825}{30134} & \frac{2043}{30134} & \frac{203}{15067} & \frac{2}{61} & -\frac{1002}{15067} & -\frac{32}{247} & -\frac{1743}{15067} & \frac{17577}{30134} & \frac{1932}{15067} & -\frac{27237}{30134} & -\frac{15645}{30134} \\
\frac{514}{15067} & -\frac{825}{30134} & -\frac{2043}{30134} & -\frac{203}{15067} & -\frac{2}{61} & \frac{1002}{15067} & \frac{32}{247} & \frac{1743}{15067} & \frac{12557}{30134} & -\frac{1932}{15067} & -\frac{2897}{30134} & -\frac{14489}{30134} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \tag{35}$$

Again, we would like to formulate a more compact solution in form of a 12×3 matrix $\underline{\underline{M_2}}$ so that

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \end{pmatrix} = \underline{\underline{M_2}} \begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \end{pmatrix} \tag{36}$$

This matrix $\underline{\underline{M_2}}$ can be derived from solution (34) and (35). When setting $Q_1 = 1$, $Q_2 = 0$, and $\overline{Q_3} = 0$, (34) returns a vector representing the first column of matrix M_2. Similarly, setting $Q_1 = 0$, $Q_2 = 1$, and $Q_3 = 0$ produces the second column of matrix $\underline{\underline{M_2}}$; and when we set $Q_1 = 0$, $Q_2 = 0$, and $Q_3 = 1$ we get the third column of matrix $\underline{\underline{M_2}}$.

Hence, the final and most compact solution with regards to our special-case problem can be written as:

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \end{pmatrix} = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ \frac{2185}{7708} & -\frac{2}{47} & \frac{35}{3854} \\ \frac{31198049}{116136436} & -\frac{74}{2867} & \frac{208413}{29034109} \\ \frac{22982883}{116136436} & \frac{196}{2867} & -\frac{944171}{58068218} \\ \frac{21441469}{232272872} & \frac{2349}{11468} & -\frac{10949897}{232272872} \\ \frac{19207}{12224888} & \frac{3385}{11468} & -\frac{571395}{12224888} \\ -\frac{571395}{12224888} & \frac{3385}{11468} & \frac{19207}{12224888} \\ -\frac{10949897}{232272872} & \frac{2349}{11468} & \frac{21441469}{232272872} \\ -\frac{944171}{58068218} & \frac{196}{2867} & \frac{22982883}{116136436} \\ \frac{208413}{29034109} & -\frac{74}{2867} & \frac{31198049}{116136436} \\ \frac{35}{3854} & -\frac{2}{47} & \frac{2185}{7708} \\ 0 & 0 & \frac{1}{4} \end{bmatrix} \begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \end{pmatrix} \tag{37}$$

Based on this descriptive toy-example, we can now easily deduct the algorithm for a general $\underline{\underline{M_2}}$ matrix of arbitrary size:

(1) Let Q_j be the single original charge in the j-th Wigner-Seitz cell and j_{max} the total number of cells. Further, let n be the number of (smaller) sub-charges to be created for each original charge Q_j. Then $m = nj_{max}$ is the total number of smaller sub-charges $q_1 \ldots q_m$. In the first step, we calculate matrix $\underline{\underline{M_1}}$ as explained in chapter 3.2.1.

(2) Then, we generate a Matrix $\underline{\underline{B}}$ of size $m \times m$ with elements $b_{i,j}$, and we set all matrix elements $b_{i,j} = 0$.

(3) In each $(kn+1)$-th line of matrix $\underline{\underline{B}}$ (with $k \in \mathbb{N}_{\geq 0}$), we set $b_{kn+1,kn+1} = 1$, $b_{kn+1,kn+2} = 1$, $b_{kn+1,kn+3} = 1$, and $b_{kn+1,kn+4} = 1$ (this corresponds to the red entries in the matrix equation (33)).

(4) We then set the first entry in the second line $b_{2,1} = 1$ and the bottom-right entry $b_{m,m} = 1$ (this corresponds to the blue entries in the matrix equation (33)).

(5) We also set $b_{3,2} = 1$ and $b_{m-1,m-1} = 1$ (this corresponds to the brown entries in the matrix equation (33)).

(6) In all lines i hitherto not yet modified, we set $b_{i,i-3} = \frac{1}{2}$, $b_{i,i-2} = -\frac{5}{2}$, $b_{i,i-1} = 5$, $b_{i,i} = -5$, $b_{i,i+1} = \frac{5}{2}$, and $b_{i,i+2} = -\frac{1}{2}$ (this corresponds to the black entries in the matrix equation (33)).

(7) We calculate $\underline{\underline{B}}^{-1}$ by inverting matrix $\underline{\underline{B}}$.

(8) We generate a row-vector $\vec{Q}$ with $j_{max}N$ lines, and we set all entries to zero, except for every $(kn+1)$-th entry (with $k \in \mathbb{N}_{\geq 0}$), which we define to be Q_{k+1}. Additionally, we define the second entry to be $\frac{Q_1}{n}$, and the last entry to be $\frac{Q_{jmax}}{n}$.

(9) We calculate $\tilde{q}_2$ and $\tilde{q}_{m-1}$ from the following equation, leaving $Q_1 \ldots Q_{j_{max}}$ as abstract variables.

$$\begin{pmatrix} \tilde{q}_1 \\ \tilde{q}_2 \\ \vdots \\ \tilde{q}_{m-1} \\ \tilde{q}_m \end{pmatrix} = \underline{\underline{M_1}} \begin{pmatrix} Q_1 \\ \vdots \\ Q_{j_{max}} \end{pmatrix} \tag{38}$$

Then, we set the third entry of the previously generated row-vector $\vec{Q}$ to be $\tilde{q}_2$, and the second last entry to be $\tilde{q}_{m-1}$.

(10) Let $\vec{m}_j$ be the j-th column of matrix $\underline{\underline{M_2}}$. We create Matrix $\underline{\underline{M_2}}$ by calculating all columns $\vec{m}_j$ according to the following rule: $\vec{m}_j = \underline{\underline{B}}^{-1}\vec{Q}|_{Q_i=1 \; if \; i=j \; else \; 0}$

3.3 Implementation in C++

We implemented the algorithm presented in chapter 3.2 in an easy-to-use, encapsulated C++ class. The complete source code listing can be found in Appendix 6.3.

All functionality is encapsulated in class `clsChargeDistr`. The number of original (coarse) charges $Q_1 \ldots Q_n$, and the so-called "split-factor" n (i.e. the number of sub-charges to be created for each original charge) can already be passed to the class constructor, which then calculates matrices $\underline{\underline{M_1}}$ and $\underline{\underline{M_2}}$ as described in chapter 3.2.

The values of the original charges $Q_1 \ldots Q_n$ can be passed to the class instance either by means of an C++ array of type `double`, or by a `vector<double>` object. Alternatively, each charge value Q_j can be set individually.

Similarly, the set of refined charges $q_1 \ldots q_m$ can also be read out by means of an C++ array of type `double`, or in a `vector<double>` object. Alternatively, each refined sub-charge value q_i can be read out individually.

3.3.1 Passing and Retrieving Charges in C++ Arrays

The following source code listing shows how to pass original charge values to and get refined charge values from the class instance by means of C++ arrays of type `double`:

```
   #include <iostream>
   #include "clsChargeDistr.h"

   using namespace std;
 5
   int main(int argc, char *argv[])
   {
     double ChargeArr[8];
     ChargeArr[0] = 0.13714240;
10   ChargeArr[1] = 0.63355233;
     ChargeArr[2] = 0.66643012;
     ChargeArr[3] = 0.41865784;
     ChargeArr[4] = 0.57789615;
     ChargeArr[5] = 0.79494448;
15   ChargeArr[6] = 0.25404124;
     ChargeArr[7] = 0.17435211;

     clsChargeDistr chargeRefiner(8, 4); // init: 8 charges to be divided into 4 subcharges each
     chargeRefiner.setChargeArray(ChargeArr); // pass array with original charges
20
     cout << "Number of original charges: " << chargeRefiner.getChrgCount() << endl;
     cout << "Number of refined charges: " << chargeRefiner.getDistrChrgCount() << endl << endl;

     cout << "Original Charge values: " << endl;
25   for (int i = 0; i < chargeRefiner.getChrgCount(); i++)
       cout << i + 1 << ": " << chargeRefiner.getSingleCharge(i) << ";  ";
     cout << endl << endl;

     cout << "Refined charges with constant SECOND derivate: " << endl;
30   double* solution;
     solution = chargeRefiner.getRefinedChargeArray(1); // get solution array (const 2nd deriv)
     for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
     {
       cout << i + 1 << ": " << solution[i] << ";  ";
35     if ((i+1) % 7 == 0) cout << endl; // add CR every 7 numbers
     }
     cout << endl << endl;

     cout << "Refined charges with constant FOURTH derivate: " << endl;
40   solution = chargeRefiner.getRefinedChargeArray(2); // get solution array (const 4th deriv)
     for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
     {
       cout << i + 1 << ": " << solution[i] << ";  ";
       if ((i + 1) % 7 == 0) cout << endl; // add CR every 7 numbers
45   }
     return(0);
   }
```

In lines 8-16, the original charge values $Q_1 \ldots Q_8$ (with index numbers $0 \cdots 7$) are defined in an C++ array of type `double`. In line 18, an instance of `clsChargeDistr` is created and initialized to handle 8 original charges, each of which are to be split into 4 sub-charges. In line 19, the array with the original charges is passed to the instance. Lines 21-27 just demonstrate how information about the original charge distribution can be retrieved from the class instance.

Lines 29-37 show, how by calling method `getRefinedChargeArray(1)`, the refined charges $q_1 \ldots q_{32}$ with constant *second* derivative can be retrieved from the class instance (by reference) in a simple C++-array (index numbers $0 \ldots 31$), and then printed. Line 35 just ads an carriage-return character every seven numbers.

Similarly, in lines 39-45, by calling method `getRefinedChargeArray(2)`, the refined charges $q_1 \ldots q_{32}$ with constant *fourth* derivative are retrieved and printed.

Below is the output created by this short program:

```
Number of original charges: 8
Number of refined charges: 32

Original Charge values:
1: 0.137142;  2: 0.633552;  3: 0.66643;  4: 0.418658;  5: 0.577896;  6: 0.794944;  7: 0.254041;  8: 0.174352;

Refined charges with constant SECOND derivate:
1: 0.0342856;  2: 0.0185916;  3: 0.0264386;  4: 0.0578266;  5: 0.112756;  6: 0.15298;  7: 0.1785;
8: 0.189316;  9: 0.185427;  10: 0.176344;  11: 0.162066;  12: 0.142593;  13: 0.117927;  14: 0.102755;
15: 0.0970787;  16: 0.100898;  17: 0.114212;  18: 0.131643;  19: 0.153189;  20: 0.178852;  21: 0.208631;
22: 0.216585;  23: 0.202713;  24: 0.167016;  25: 0.109493;  26: 0.0680912;  27: 0.0428092;  28: 0.0336474;
29: 0.0406058;  30: 0.0445821;  31: 0.0455761;  32: 0.043588;

Refined charges with constant FOURTH derivate:
1: 0.0342856;  2: 0.0185916;  3: 0.0263434;  4: 0.0579218;  5: 0.104915;  6: 0.150119;  7: 0.182304;
8: 0.196215;  9: 0.192575;  10: 0.178076;  11: 0.158262;  12: 0.137517;  13: 0.119075;  14: 0.105013;
15: 0.0972125;  16: 0.0973572;  17: 0.106934;  18: 0.127234;  19: 0.156049;  20: 0.187679;  21: 0.212923;
22: 0.219087;  23: 0.201153;  24: 0.161781;  25: 0.111309;  26: 0.0677507;  27: 0.0411925;  28: 0.0337895;
29: 0.0397671;  30: 0.0454209;  31: 0.0455761;  32: 0.043588;
```

3.3.2 Passing and Retrieving Charges in Vectors

The following source code listing shows how to pass original charge values to and get refined charge values from the class instance by means of objects of type `vector<double>`:

```cpp
#include <iostream>
#include <vector>
#include "clsChargeDistr.h"

using namespace std;

int main(int argc, char *argv[])
{
  vector<double> ChargeVec;
  ChargeVec.push_back(0.13714240);
  ChargeVec.push_back(0.63355233);
  ChargeVec.push_back(0.66643012);
  ChargeVec.push_back(0.41865784);
  ChargeVec.push_back(0.57789615);
  ChargeVec.push_back(0.79494448);
  ChargeVec.push_back(0.25404124);
  ChargeVec.push_back(0.17435211);

  clsChargeDistr chargeRefiner(8, 4); // init: 8 charges to be divided into 4 subcharges each
  chargeRefiner.setChargeVector(ChargeVec); // pass vector with original charges

  cout << "Number of original charges: " << chargeRefiner.getChrgCount() << endl;
  cout << "Number of refined charges: " << chargeRefiner.getDistrChrgCount() << endl << endl;

  cout << "Original Charge values: " << endl;
  for (int i = 0; i < chargeRefiner.getChrgCount(); i++)
    cout << i + 1 << ": " << chargeRefiner.getSingleCharge(i) << ";  ";
  cout << endl << endl;

  cout << "Refined charges with constant SECOND derivate: " << endl;
  vector<double> solution;
  solution = chargeRefiner.getRefinedChargeVector(1); // get solution vector (const 2nd deriv)
  for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
  {
    cout << i + 1 << ": " << solution[i] << ";  ";
    if ((i+1) % 7 == 0) cout << endl; // add CR every 7 numbers
  }
  cout << endl << endl;

  cout << "Refined charges with constant FOURTH derivate: " << endl;
  solution = chargeRefiner.getRefinedChargeVector(2); // get solution array (const 4th deriv)
  for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
  {
    cout << i + 1 << ": " << solution[i] << ";  ";
    if ((i + 1) % 7 == 0) cout << endl; // add CR every 7 numbers
  }
  return(0);
}
```

This program has the same functionality as the one before. The only differences are that it uses method `setChargeVector` in line 20, and method `getRefinedChargeVector` in lines 33 and 41.

3.3.3 Passing and Retrieving Charges One by One

The following source code listing shows how to pass original charge values Q_j one by one to the class instance, and how to get refined charge values q_i back one by one from the class instance:

```
   #include <iostream>
   #include "clsChargeDistr.h"

   using namespace std;
 5
   int main(int argc, char *argv[])
   {
     clsChargeDistr chargeRefiner(8, 4); // init: 8 charges to be divided into 4 subcharges each

10   // pass values of original charges one by one
     chargeRefiner.setSingleCharge(0, 0.13714240);
     chargeRefiner.setSingleCharge(1, 0.63355233);
     chargeRefiner.setSingleCharge(2, 0.66643012);
     chargeRefiner.setSingleCharge(3, 0.41865784);
15   chargeRefiner.setSingleCharge(4, 0.57789615);
     chargeRefiner.setSingleCharge(5, 0.79494448);
     chargeRefiner.setSingleCharge(6, 0.25404124);
     chargeRefiner.setSingleCharge(7, 0.17435211);

20   cout << "Number of original charges: " << chargeRefiner.getChrgCount() << endl;
     cout << "Number of refined charges: " << chargeRefiner.getDistrChrgCount() << endl << endl;

     cout << "Original Charge values: " << endl;
     for (int i = 0; i < chargeRefiner.getChrgCount(); i++)
25     cout << i + 1 << ": " << chargeRefiner.getSingleCharge(i) << ";  ";
     cout << endl << endl;

     cout << "Refined charges with constant SECOND derivate: " << endl;
     for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
30   {
       // get i-th refined charge (const 2nd derivative)
       cout << i + 1 << ": " << chargeRefiner.getRefinedCharge(1,i) << ";  ";
       if ((i+1) % 7 == 0) cout << endl; // add CR every 7 numbers
     }
35   cout << endl << endl;

     cout << "Refined charges with constant FOURTH derivate: " << endl;
     for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
     {
40     // get i-th refined charge (const 4th derivative)
       cout << i + 1 << ": " << chargeRefiner.getRefinedCharge(2, i) << ";  ";
       if ((i + 1) % 7 == 0) cout << endl; // add CR every 7 numbers
     }
     return(0);
45 }
```

Again, this program has the same functionality as the two versions before. The only differences are that it uses method `setSingleCharge` in lines 10-18, and method `getRefinedCharge` in lines 32 and 41.

3.3.4 Retrieving Matrices M_1 and M_2

The following code example demonstrates how to retrieve the complete $\underline{\underline{M_1}}$ and $\underline{\underline{M_2}}$ matrix from a class instance:

```cpp
#include <iostream>
#include "clsChargeDistr.h"
using namespace std;

int main(int argc, char *argv[])
{
  clsChargeDistr chargeRefiner(8, 4); // init: 8 charges to be divided into 4 subcharges each

  // Print matrix M1
  cout.setf(ios::fixed, ios::floatfield);
  cout << " MATRIX M1: " << endl;
  for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
  {
    for (int j = 0; j < chargeRefiner.getChrgCount(); j++)
    {
      if (chargeRefiner.getM1cell(i, j) >= 0) cout << " "; // print extra space if >=0
      cout << chargeRefiner.getM1cell(i, j) << " "; // print M1(i,j)
    }
    cout << endl;
  }
  cout << endl << endl;

  // Print matrix M2
  cout << " MATRIX M2: " << endl;
  for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
  {
    for (int j = 0; j < chargeRefiner.getChrgCount(); j++)
    {
      if (chargeRefiner.getM2cell(i, j) >= 0) cout << " "; // print extra space if >=0
      cout << chargeRefiner.getM2cell(i, j) << " "; // print M2(i,j)
    }
    cout << endl;
  }
  return(0);
}
```

Below is the output created by this short program:

```
 MATRIX M1:
 0.250000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
 0.283171 -0.041146  0.009892 -0.002378  0.000572 -0.000138  0.000034 -0.000007
 0.266586 -0.020573  0.004946 -0.001189  0.000286 -0.000069  0.000017 -0.000004
 0.200243  0.061719 -0.014838  0.003567 -0.000858  0.000207 -0.000051  0.000011
 0.084143  0.205731 -0.049459  0.011891 -0.002859  0.000689 -0.000171  0.000036
 0.004046  0.281043 -0.043524  0.010464 -0.002516  0.000606 -0.000150  0.000032
-0.040049  0.287656  0.002968 -0.000713  0.000172 -0.000041  0.000010 -0.000002
-0.048140  0.225570  0.090016 -0.021641  0.005203 -0.001253  0.000311 -0.000066
-0.020229  0.094786  0.217621 -0.052318  0.012579 -0.003030  0.000751 -0.000160
-0.000973  0.004557  0.281615 -0.043662  0.010498 -0.002528  0.000627 -0.000134
 0.009628 -0.045114  0.281997  0.004328 -0.001041  0.000251 -0.000062  0.000013
 0.011573 -0.054229  0.218767  0.091652 -0.022036  0.005307 -0.001316  0.000281
 0.004863 -0.022787  0.091927  0.218310 -0.052489  0.012641 -0.003134  0.000669
 0.000234 -0.001096  0.004420  0.281649 -0.043674  0.010519 -0.002607  0.000556
-0.002315  0.010846 -0.043753  0.281670  0.004407 -0.001061  0.000263 -0.000056
-0.002782  0.013037 -0.052593  0.218372  0.091756 -0.022099  0.005478 -0.001169
-0.001169  0.005478 -0.022099  0.091756  0.218372 -0.052593  0.013037 -0.002782
-0.000056  0.000263 -0.001061  0.004407  0.281670 -0.043753  0.010846 -0.002315
 0.000556 -0.002607  0.010519 -0.043674  0.281649  0.004420 -0.001096  0.000234
 0.000669 -0.003134  0.012641 -0.052489  0.218310  0.091927 -0.022787  0.004863
 0.000281 -0.001316  0.005307 -0.022036  0.091652  0.218767 -0.054229  0.011573
 0.000013 -0.000062  0.000251 -0.001041  0.004328  0.281997 -0.045114  0.009628
-0.000134  0.000627 -0.002528  0.010498 -0.043662  0.281615  0.004557 -0.000973
-0.000160  0.000751 -0.003030  0.012579 -0.052318  0.217621  0.094786 -0.020229
-0.000066  0.000311 -0.001253  0.005203 -0.021641  0.090016  0.225570 -0.048140
-0.000002  0.000010 -0.000041  0.000172 -0.000713  0.002968  0.287656 -0.040049
 0.000032 -0.000150  0.000606 -0.002516  0.010464 -0.043524  0.281043  0.004046
 0.000036 -0.000171  0.000689 -0.002859  0.011891 -0.049459  0.205731  0.084143
 0.000011 -0.000051  0.000207 -0.000858  0.003567 -0.014838  0.061719  0.200243
-0.000004  0.000017 -0.000069  0.000286 -0.001189  0.004946 -0.020573  0.266586
-0.000007  0.000034 -0.000138  0.000572 -0.002378  0.009892 -0.041146  0.283171
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.250000
 MATRIX M2:
 0.250000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
 0.283171 -0.041146  0.009892 -0.002378  0.000572 -0.000138  0.000034 -0.000007
 0.267449 -0.022894  0.007714 -0.003222  0.001352 -0.000572  0.000257 -0.000085
 0.199380  0.064040 -0.017606  0.005600 -0.001924  0.000710 -0.000291  0.000092
 0.098603  0.188558 -0.050775  0.018940 -0.007467  0.003042 -0.001336  0.000435
 0.007850  0.279179 -0.050140  0.018380 -0.007431  0.003084 -0.001370  0.000449
-0.047423  0.296928  0.001921 -0.001950  0.000714 -0.000264  0.000109 -0.000034
-0.059030  0.235335  0.098994 -0.035371  0.014184 -0.005862  0.002598 -0.000849
-0.036061  0.120436  0.210630 -0.062652  0.024884 -0.010324  0.004591 -0.001503
-0.004879  0.010768  0.282347 -0.052013  0.019401 -0.008027  0.003573 -0.001171
 0.017506 -0.058021  0.286206  0.007692 -0.004817  0.002047 -0.000909  0.000297
 0.023434 -0.073183  0.220817  0.106973 -0.039468  0.016305 -0.007254  0.002376
 0.014597 -0.044861  0.111334  0.215708 -0.065390  0.026547 -0.011801  0.003866
 0.002038 -0.006094  0.009457  0.283139 -0.052564  0.019963 -0.008830  0.002892
-0.007094  0.021788 -0.053614  0.283772  0.008949 -0.005478  0.002496 -0.000819
-0.009541  0.029167 -0.067177  0.217381  0.109005 -0.041032  0.018136 -0.005939
-0.005939  0.018136 -0.041032  0.109005  0.217381 -0.067177  0.029167 -0.009541
-0.000819  0.002496 -0.005478  0.008949  0.283772 -0.053614  0.021788 -0.007094
 0.002892 -0.008830  0.019963 -0.052564  0.283139  0.009457 -0.006094  0.002038
 0.003866 -0.011801  0.026547 -0.065390  0.215708  0.111334 -0.044861  0.014597
 0.002376 -0.007254  0.016305 -0.039468  0.106973  0.220817 -0.073183  0.023434
 0.000297 -0.000909  0.002047 -0.004817  0.007692  0.286206 -0.058021  0.017506
-0.001171  0.003573 -0.008027  0.019401 -0.052013  0.282347  0.010768 -0.004879
-0.001503  0.004591 -0.010324  0.024884 -0.062652  0.210630  0.120436 -0.036061
-0.000849  0.002598 -0.005862  0.014184 -0.035371  0.098994  0.235335 -0.059030
-0.000034  0.000109 -0.000264  0.000714 -0.001950  0.001921  0.296928 -0.047423
 0.000449 -0.001370  0.003084 -0.007431  0.018380 -0.050140  0.279179  0.007850
 0.000435 -0.001336  0.003042 -0.007467  0.018940 -0.050775  0.188558  0.098603
 0.000092 -0.000291  0.000710 -0.001924  0.005600 -0.017606  0.064040  0.199380
-0.000085  0.000257 -0.000572  0.001352 -0.003222  0.007714 -0.022894  0.267449
-0.000007  0.000034 -0.000138  0.000572 -0.002378  0.009892 -0.041146  0.283171
 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.250000
```

3.3.5 Generating Matrix Files by Command Line Parameter Calls

The program shown on the following page, named `MatrixGen`, will be used in chapter 4. It accepts command line parameters, and generates a text file containing matrix $\underline{\underline{M_1}}$ or $\underline{\underline{M_2}}$ for arbitrary charge distributions and split factors. This is its usage:

```
MatrixGen Qcount SplitFactor MatrixType
```

`Qcount`	number of original (coarse) charges
`SplitFactor` ...	number of original (coarse) charges
`MatrixType`	1 for matrix $\underline{\underline{M_1}}$, or 2 for $\underline{\underline{M_2}}$

The name of the generated file is determined by the command line parameters. If e.g. the program is called with `MatrixGen 8 4 2`, then a file named `Matrix2_8_4.txt` is generated.

```
#include <iostream>
#include <fstream>
#include "clsChargeDistr.h"
#include <sstream>
#include <string>
using namespace std;

// ************************************
// Charge Distribution Matrix Generator
// 1st param: number of charges
// 2nd param: split factor
// 3rd param: 1 or 2 for M1 or M2
// ************************************
int GenerateMatrix(int, int, int);
int main(int argc, char *argv[])
{
  if (argc < 4)
  {
    cerr << "You must provide 3 parameters ";
    cerr << "(number of charges, split factor and matrix type (1 or 2))" << endl;
    return(1);
  }

  int QCount;
  stringstream ss1;
  ss1 << argv[1];
  ss1 >> QCount;

  int SplitFactor;
  stringstream ss2;
  ss2 << argv[2];
  ss2 >> SplitFactor;

  int MatrixType;
  stringstream ss3;
  ss3 << argv[3];
  ss3 >> MatrixType;

  return(GenerateMatrix(QCount,SplitFactor, MatrixType));
}

// *****************************
// Generate Matrix
// *****************************
int GenerateMatrix(int ChargeCount, int SplitFactor, int MatrixType)
{
  clsChargeDistr chargeRefiner;
  fstream MatrixFile;
  if (!chargeRefiner.Prepare(ChargeCount, SplitFactor))
    return(1);

  stringstream filename;
  if (MatrixType != 1) MatrixType = 2;
  filename << "Matrix" << MatrixType << "_" << ChargeCount << "_" << SplitFactor << ".txt";

  MatrixFile.open(filename.str(), ios_base::out);
  if (!MatrixFile)
  {
    cerr << "Error creating File " << filename.str() << endl;
    return(1);
  }

  typedef std::numeric_limits< double > dbl;
  MatrixFile.precision(dbl::max_digits10 + 2);
  for (int i = 0; i < chargeRefiner.getDistrChrgCount(); i++)
  {
    if(i>0) MatrixFile << endl;
    for (int j = 0; j < chargeRefiner.getChrgCount(); j++)
    {
      if (j > 0) MatrixFile << ", ";
      if(MatrixType==1)
        MatrixFile << chargeRefiner.getM1cell(i, j);
      else
        MatrixFile << chargeRefiner.getM2cell(i, j);
    }
  }
  MatrixFile.close();
  return(0);
}
```

3.3.6 Explanations to the Charge Refinement Class Core Function

On page 77ff of chapter 6.3.2, the source code of method `Prepare(int, int)` is listed. All line numbers in this chapter refer to this listing. This method is the core function of this class, as it calculates matrices $\underline{\underline{M_1}}$ and $\underline{\underline{M_2}}$ according to the algorithm explained in chapter 2.1.1. The method `Prepare(int, int)` can be called explicitly (e.g. if the class has been instantiated with the empty constructor, or if one wishes to change parameters), but it is also utilized by the constructor `clsChargeDistr(int NoOfCharges, int SplitFactor)`.

In line 196, all former calculations of refined charges are flagged as invalid. In lines 198-209, some plausibility checks are performed. In lines 211-232, additional memory is allocated, if necessary, for the arrays holding the original charges, and the refined charges. Eventually, in lines 233-234, the array holding the original charge values is zeroed, and (in lines 236-237) the `NoOfCharges` and `SplitFactor` parameters are stored as the now current parameters.

Calculation of matrix $\underline{\underline{M_1}}$ starts in line 239. In lines 242-246, an empty matrix $\underline{\underline{A}}$ is created and filled with zeros (corresponding to step (1) on page 22). The matrix object is from the "Eigen"-library) **http://eigen.tuxfamily.org/** (Version 3.3.7). This library is also used for all matrix/vector operations and for matrix inversion. It is included in line 5 of the header file.

In the loop starting from line 247, all matrix elements of matrix $\underline{\underline{A}}$ are created. Lines 250-255 correspond to step (2) on page 22, lines 256-267 correspond to step (3) on page 22, and lines 268-275 correspond to step (4) on page 22.

In lines 278-281, the inverted matrix $\underline{\underline{A}}^{-1}$ is calculated (utilizing the "Eigen"-library). This corresponds to step (5) on page 22. As some of the entries of $\underline{\underline{A}}^{-1}$ are bound to be zero, these entries are overwritten with zeros in lines 283-290. This is to minimize small deviations due to numerical artifacts of the matrix inversion algorithm.

Eventually, in lines 293 and 294, an empty matrix $\underline{\underline{M_1}}$ is created and a vector $\vec{Q}$ is defined. In the loop starting in line 295, a row-vector $\vec{Q}$ and a column vector $\vec{m}_j$ of the final matrix is calculated and added to $\underline{\underline{M_1}}$ in each pass, corresponding to steps (6) and (7) on page 22. This concludes calculation of matrix $\underline{\underline{M_1}}$.

Starting from line 309, matrix $\underline{\underline{M_2}}$ is calculated. Corresponding to step (2) on page 26, an empty, all-zero matrix is created in line 313. Please note that on page 26 this matrix is called $\underline{\underline{B}}$, whereas in the source-code, variable `A` is re-used.

In the loop starting from line 314, all matrix elements of matrix $\underline{\underline{B}}$ (source code variable `A`) are created. Lines 316-222 correspond to step (3) on page 26, lines 256-267 and 341-346 correspond to step (4) on page 26, lines 329-340 correspond to step (5) on page 26, and lines 347-356 correspond to step (6) on page 26.

In line 360, the inverted matrix $\underline{\underline{B}}^{-1}$ (source code variable `Ainv`) is calculated (again utilizing the "Eigen"-library). This corresponds to step (7) on page 26. As before, some of the entries of $\underline{\underline{B}}^{-1}$ (source code variable `Ainv`) are bound to be zero and are overwritten with zeros in lines 362-378. This is, again, to minimize small deviations due to numerical artifacts of the matrix inversion algorithm.

Eventually, in line 381, an empty matrix $\underline{\underline{M_2}}$ is created. In the loop starting in line 383, a row-vector $\vec{Q}$ and a column vector $\vec{m}_j$ of the final matrix is calculated and added to $\underline{\underline{M_2}}$ in each pass, corresponding to steps (8), (9) and (10) on page 27. This concludes calculation of matrix $\underline{\underline{M_2}}$.

4 Back to Neural Networks

4.1 Can Matrix M$_2$ be Retrieved From a Trained Neural Network?

When you compare the exact solution for matrix $\underline{\underline{M_2}}$ on page 32 with matrices $\underline{\underline{W_1}}$ and $\underline{\underline{W_2}}$ on page 17, as extracted from a trained neural network, you will find no obvious similarity. This is not surprising, though. Matrices $\underline{\underline{W_1}}$ and $\underline{\underline{W_2}}$, together with vectors $\vec{b}_1$ and $\vec{b}_2$, encode the solution redundantly, and also the network having produced these matrices was trained with scaled target charges.

But scaling was used during the initial experimental phase only in consideration of the limited output value range of various non-linear activation functions. The final network employs just the linear activation function. Its output neurons can, in principle, produce any positive or negative value, and therefore scaling is not required. Hence, equation (17) already represents the output vector of such network.

By eliminating the brackets, equation (17) can be re-written as

$$\vec{q} = \underline{\underline{W_2}}\;\underline{\underline{W_1}}\;\vec{Q} + \underline{\underline{W_2}}\;\vec{b}_1 + \vec{b}_2 \tag{39}$$

By comparing this equation with equation (36), we see that - if the network produces a solution equivalent to (36) - the following two relations must be true:

$$\underline{\underline{W_2}}\;\underline{\underline{W_1}} = \underline{\underline{M_2}} \tag{40}$$

$$\underline{\underline{W_2}}\;\vec{b}_1 + \vec{b}_2 = \vec{0} \tag{41}$$

This was checked with the following program:

```
"""
CR_conv13.py
Charge Distribution Effective Matrix Calculator
BY HELMUT HOERNER
(C) 2019
"""
import os
import numpy as np
from keras import models
from keras import layers
from keras import callbacks

QCount=8;
SplitFactor=4;
FileName = 'Matrix2_'+str(QCount)+'_'+str(SplitFactor)+'.txt'
np.random.seed(0) # make pseudo random numbers reproducible
maxepochs=10000 # max number of epochs
mypat=20 # stop after this no of epochs if no improvement
trainSetSize=1500000 # training data
valSetSize  = 300000 # validation data
bSize=150000 # batch Size
myoptimizer='Adam' # optimizer

# ********************************************
# Helper function for printing a Matrix
# ********************************************
def printMatrix(M_name, M):
    print("")
    print("***************")
    print(M_name)
    print("***************")
    for row in M:
        for val in row:
            if val>=0:
                print (' ',end='')
            print("%2.6f" % val, end=' ')
        print("")

# ********************************************
# Helper function for printing a Vector
# ********************************************
def printVector(V_name, V):
    print("")
    print("***************")
    print(V_name)
    print("***************")
    for val in V:
        if val>=0:
            print (' ',end='')
        print("%2.6f" % val)


# ********************************************
# Generate and load matrix M2
# ********************************************
if os.path.isfile(FileName):
    print(FileName,"already exists.")
else:
    print('Generating matrix file',FileName)
    os.system('MatrixGen '+str(QCount)+' '+str(SplitFactor)+' 2')

print('Load matrix file')
f=open(FileName,encoding="utf-8")
MatrixData=f.read()
f.close()
MatrixLines=MatrixData.split('\n')
Qcount = len(MatrixLines[0].split(','))
qcount = len(MatrixLines)
SplitFactor = int(qcount/Qcount)

# now parsing matrix data
M2 = np.zeros((qcount, Qcount))
for i, line in enumerate(MatrixLines):
    sline=line.split(',')
    values = [float(x) for x in sline]
    M2[i,:]=values
```

```
# **********************************
# Charge Generator
# **********************************
def chrg_generator(n):
    """ Creates test set with n entries """
    for i in range(n):
        # Creates random full charges
        Q=np.random.rand(Qcount)
        q=M2@Q
        yield([i-1, q, Q])

# *************************************
# Generate data
# *************************************
totSize=trainSetSize+valSetSize
data=np.zeros((totSize,Qcount))
targets=np.zeros((totSize,qcount))

offset=0
for i, q, Q in chrg_generator(totSize):
    if i%100000==0:
        print("Computing TrainSet",i,"-",i+99999)
    data[i]=Q
    targets[i]=q

# Separate Train Data
train_data=data[:trainSetSize]
train_targets=targets[:trainSetSize]

# Separate Validation Data
val_data=data[trainSetSize:trainSetSize+valSetSize]
val_targets=targets[trainSetSize:trainSetSize+valSetSize]

callbacks_list = [
        callbacks.EarlyStopping(monitor='val_loss', patience=mypat,)
        ]

# **********************************
# Create and train Model
# **********************************
print()
print("**************")
print("Train model")
print("**************")
model=models.Sequential()
model.add(layers.Dense(Qcount, activation='linear',
                               input_dim=train_data.shape[1]))
model.add(layers.Dense(qcount))
model.summary()
model.compile(optimizer=myoptimizer, loss='mse', metrics=['mae'])
history=model.fit(train_data,train_targets,
                      epochs=maxepochs, batch_size=bSize,
                      callbacks=callbacks_list,
                      validation_data=(val_data,val_targets))

# ********************************************
# Print effective Matrix and vector
# ********************************************
W1=np.transpose(model.get_weights()[0]) # weight matrix input layer
B1=np.transpose(model.get_weights()[1]) # bias vector input layer
W2=np.transpose(model.get_weights()[2]) # weight matrix output layer
B2=np.transpose(model.get_weights()[3]) # bias vector output layer
printMatrix("MATRIX W2.W1",W2@W1)
printVector("VECTOR W2.B1+B2",W2@B1+B2)
```

The above program trains a dense network for an initial charge distribution of 8 charges (line 13), to be split in 4 charges each (line 14). The number of epochs after which the training should stop if there is no improvement is set to 20 (line 18). Total training set size is 1,500,000, with a validation set of size 300,000 (lines 19 and 20). Batch size is chosen to be 150000 (line 21), and this time we use the `Adam` optimizer (line 23), which has proven in the initial experimentation phase to be very effective for these linear dense networks.

Lines 24-50 just encode two helper function for printing out matrices and vectors. Starting from line 32, a matching $\underline{\underline{M_2}}$ matrix file is generated (if not already available) by calling the `MatrixGen`-program from chapter 3.3.5, and then matrix $\underline{\underline{M_2}}$ is loaded for further use.

Eventually, in lines 78-109, data sets for training and validating are generated at high speed by utilizing the $\underline{\underline{M_2}}$ matrix. Starting from line 115, a simple dense network with just one 8-neuron input layer, and one $8 \times 4 = 32$ neuron output layer is created and trained. In lines 134-141, matrix $\underline{\underline{W_2}}\ \underline{\underline{W_1}}$ and vector $\underline{\underline{W_2}}\ \vec{b}_1 + \vec{b}_2$ are printed.

This is the output generated by the program:

```
***************
PROGRAM STARTED
***************
Matrix2_8_4.txt already exists.
Load matrix file
Computing TrainSet 0 - 999
Computing TrainSet 100000 - 199999
Computing TrainSet 200000 - 299999
Computing TrainSet 300000 - 399999
Computing TrainSet 400000 - 499999
Computing TrainSet 500000 - 599999
Computing TrainSet 600000 - 699999
Computing TrainSet 700000 - 799999
Computing TrainSet 800000 - 899999
Computing TrainSet 900000 - 999999
Computing TrainSet 1000000 - 1099999
Computing TrainSet 1100000 - 1199999
Computing TrainSet 1200000 - 1299999
Computing TrainSet 1300000 - 1399999
Computing TrainSet 1400000 - 1499999
Computing TrainSet 1500000 - 1599999
Computing TrainSet 1600000 - 1699999
Computing TrainSet 1700000 - 1799999

**************
Train model
**************
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 8)                 72
_________________________________________________________________
dense_12 (Dense)             (None, 32)                288
=================================================================
Total params: 360
Trainable params: 360
Non-trainable params: 0
_________________________________________________________________
Train on 1500000 samples, validate on 300000 samples
Epoch 1/10000
1500000/1500000 [==============================] - 3s 2us/step - loss: 0.1532 - mean_absolute_error: 0.3158 -
                                                   val_loss: 0.1332 - val_mean_absolute_error: 0.2934
...

Epoch 842/10000
1500000/1500000 [==============================] - 1s 1us/step - loss: 7.0756e-09 - mean_absolute_error: 5.6289e-05 -
                                                   val_loss: 3.4702e-08 - val_mean_absolute_error: 1.4238e-04
```

```
***************
MATRIX W2.W1
***************
 0.250100  0.000083  0.000076 -0.000030  0.000130 -0.000020  0.000159  0.000065
 0.283203 -0.041110  0.009911 -0.002346  0.000601 -0.000108  0.000065  0.000013
 0.267480 -0.022869  0.007727 -0.003201  0.001378 -0.000546  0.000291 -0.000064
 0.199414  0.064109 -0.017557  0.005632 -0.001879  0.000712 -0.000245  0.000114
 0.098592  0.188547 -0.050782  0.018930 -0.007478  0.003031 -0.001352  0.000427
 0.007750  0.279043 -0.050221  0.018360 -0.007545  0.003069 -0.001514  0.000371
-0.047528  0.296766  0.001889 -0.001979  0.000585 -0.000265  0.000030 -0.000105
-0.059064  0.235306  0.098979 -0.035403  0.014154 -0.005897  0.002561 -0.000869
-0.036084  0.120414  0.210613 -0.062669  0.024855 -0.010343  0.004559 -0.001519
-0.004886  0.010761  0.282341 -0.052018  0.019392 -0.008033  0.003561 -0.001176
 0.017511 -0.058016  0.286210  0.007697 -0.004812  0.002051 -0.000904  0.000300
 0.023446 -0.073172  0.220824  0.106982 -0.039455  0.016316 -0.007237  0.002384
 0.014616 -0.044835  0.111344  0.215731 -0.065380  0.026567 -0.011775  0.003881
 0.002088 -0.006018  0.009512  0.283157 -0.052480  0.019956 -0.008730  0.002918
-0.007072  0.021807 -0.053603  0.283783  0.008971 -0.005460  0.002531 -0.000803
-0.009515  0.029190 -0.067169  0.217401  0.109021 -0.041005  0.018174 -0.005920
-0.005914  0.018161 -0.041017  0.109034  0.217402 -0.067148  0.029201 -0.009523
-0.000735  0.002620 -0.005395  0.008996  0.283861 -0.053595  0.021869 -0.007049
 0.002921 -0.008796  0.019984 -0.052529  0.283167  0.009487 -0.006063  0.002057
 0.003893 -0.011775  0.026567 -0.065369  0.215742  0.111355 -0.044829  0.014615
 0.002440 -0.007143  0.016351 -0.039414  0.107038  0.220830 -0.073121  0.023456
 0.000344 -0.000866  0.002068 -0.004831  0.007740  0.286203 -0.057977  0.017533
-0.001168  0.003576 -0.008028  0.019396 -0.052010  0.282342  0.010764 -0.004880
-0.001520  0.004570 -0.010337  0.024870 -0.062671  0.210618  0.120413 -0.036075
-0.000867  0.002584 -0.005874  0.014162 -0.035390  0.098970  0.235308 -0.059042
-0.000059  0.000085 -0.000278  0.000688 -0.001973  0.001894  0.296896 -0.047440
 0.000428 -0.001391  0.003068 -0.007450  0.018357 -0.050161  0.279149  0.007834
 0.000424 -0.001352  0.003033 -0.007485  0.018934 -0.050790  0.188546  0.098594
 0.000095 -0.000289  0.000709 -0.001927  0.005604 -0.017606  0.064047  0.199379
-0.000073  0.000264 -0.000569  0.001356 -0.003208  0.007722 -0.022877  0.267451
 0.000006  0.000045 -0.000132  0.000579 -0.002362  0.009901 -0.041126  0.283176
 0.000009  0.000006  0.000002  0.000002  0.000011  0.000005  0.000014  0.250002

***************
VECTOR W2.B1+B2
***************
 0.000072
 0.000029
 0.000025
 0.000054
-0.000014
-0.000090
-0.000121
-0.000034
-0.000023
-0.000006
 0.000009
 0.000017
 0.000022
 0.000080
 0.000027
 0.000028
 0.000031
 0.000103
 0.000031
 0.000026
 0.000045
 0.000020
-0.000009
-0.000029
-0.000030
-0.000033
-0.000027
-0.000013
 0.000003
 0.000012
 0.000015
 0.000011
```

By comparing this `W2.W1` matrix with the exact solution on page 32, it is now confirmed that the trained network indeed converges towards the exact solution. Also, vector `W2.B1+B2` is close to $\vec{0}$, as expected. After having trained the network for 843 epochs on 1,500,000 sample records, equations (40) and (41) are valid within expected deviations not larger than 0.0003 (absolute charge value).

4.2 Retrieving Matrix M$_2$ More Directly From a Neural Network

There is still room for improving efficiency, though. Obviously, it does not makes much sense to let the neural network learn bias vectors $\vec{b}_1$ and $\vec{b}_2$, as eventually they are supposed to become $\vec{0}$ anyway. So, if we train the network without bias vectors, and therefore set $\vec{b}_1 = \vec{0}$ and $\vec{b}_2 = \vec{0}$, equation (39) simplifies to

$$\vec{q} = \underline{\underline{W_2}}\ \underline{\underline{W_1}}\ \vec{Q} \tag{42}$$

The following code snippet shows the required modification in the previous program (compare with source code on page 37).

```
115 # ***********************************
    # Create and train Model
    # ***********************************
    print()
    print("**************")
120 print("Train model")
    print("**************")
    model=models.Sequential()
    model.add(layers.Dense(Qcount, activation='linear', use_bias=False,
                                    input_dim=train_data.shape[1]))
125 model.add(layers.Dense(qcount, use_bias=False))
    model.summary()
    model.compile(optimizer=myoptimizer, loss='mse', metrics=['mae'])
    history=model.fit(train_data,train_targets,
                              epochs=maxepochs, batch_size=bSize,
130                           callbacks=callbacks_list,
                              validation_data=(val_data,val_targets))

    # *********************************************
    # Print effective Matrix and vector
135 # *********************************************
    W1=np.transpose(model.get_weights()[0]) # weight matrix input layer
    W2=np.transpose(model.get_weights()[1]) # weight matrix output layer
    printMatrix("MATRIX W2.W1",W2@W1)
```

In lines 123 and 125 we have added `use_bias=False`, so that neither layer now uses a bias vector anymore. Consequently, the matrix calculation in lines 136-137 has simplified. This is the output generated by the simplified program:

```
***************
PROGRAM STARTED
***************
Matrix2_8_4.txt already exists.
Load matrix file
Computing TrainSet 0 - 999
Computing TrainSet 100000 - 199999
Computing TrainSet 200000 - 299999
Computing TrainSet 300000 - 399999
Computing TrainSet 400000 - 499999
Computing TrainSet 500000 - 599999
Computing TrainSet 600000 - 699999
Computing TrainSet 700000 - 799999
Computing TrainSet 800000 - 899999
Computing TrainSet 900000 - 999999
Computing TrainSet 1000000 - 1099999
Computing TrainSet 1100000 - 1199999
Computing TrainSet 1200000 - 1299999
Computing TrainSet 1300000 - 1399999
Computing TrainSet 1400000 - 1499999
Computing TrainSet 1500000 - 1599999
Computing TrainSet 1600000 - 1699999
Computing TrainSet 1700000 - 1799999
```

```
**************
Train model
**************
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_9 (Dense)              (None, 8)                 64
_________________________________________________________________
dense_10 (Dense)             (None, 32)                256
=================================================================
Total params: 320
Trainable params: 320
Non-trainable params: 0
_________________________________________________________________
Train on 1500000 samples, validate on 300000 samples
Epoch 1/512
Train on 1500000 samples, validate on 300000 samples
Epoch 1/10000
1500000/1500000 [==============================] - 3s 2us/step - loss: 0.1566 - mean_absolute_error: 0.3197 -
                                                   val_loss: 0.1405 - val_mean_absolute_error: 0.3021

...

Epoch 872/10000
1500000/1500000 [==============================] - 1s 1us/step - loss: 4.7060e-14 - mean_absolute_error: 1.5329e-07 -
                                                   val_loss: 2.2829e-14 - val_mean_absolute_error: 1.0929e-07

***************
MATRIX W2.W1
***************
 0.250000  0.000000  0.000000  0.000000  0.000000 -0.000000  0.000000  0.000000
 0.283171 -0.041146  0.009892 -0.002378  0.000572 -0.000138  0.000034 -0.000007
 0.267449 -0.022894  0.007714 -0.003222  0.001352 -0.000572  0.000257 -0.000084
 0.199380  0.064040 -0.017606  0.005600 -0.001924  0.000710 -0.000291  0.000092
 0.098603  0.188558 -0.050775  0.018940 -0.007467  0.003042 -0.001336  0.000435
 0.007850  0.279179 -0.050140  0.018380 -0.007431  0.003084 -0.001371  0.000449
-0.047423  0.296928  0.001921 -0.001950  0.000713 -0.000264  0.000108 -0.000034
-0.059030  0.235335  0.098994 -0.035371  0.014184 -0.005862  0.002598 -0.000849
-0.036061  0.120436  0.210630 -0.062652  0.024884 -0.010324  0.004591 -0.001503
-0.004879  0.010768  0.282347 -0.052013  0.019401 -0.008027  0.003573 -0.001171
 0.017506 -0.058021  0.286206  0.007692 -0.004817  0.002047 -0.000909  0.000297
 0.023434 -0.073183  0.220817  0.106973 -0.039468  0.016305 -0.007254  0.002376
 0.014597 -0.044861  0.111334  0.215709 -0.065390  0.026547 -0.011801  0.003866
 0.002038 -0.006094  0.009457  0.283139 -0.052564  0.019963 -0.008830  0.002892
-0.007094  0.021788 -0.053614  0.283772  0.008949 -0.005478  0.002496 -0.000819
-0.009541  0.029167 -0.067177  0.217381  0.109005 -0.041032  0.018136 -0.005939
-0.005939  0.018136 -0.041032  0.109005  0.217381 -0.067177  0.029167 -0.009541
-0.000819  0.002496 -0.005478  0.008949  0.283772 -0.053614  0.021788 -0.007094
 0.002892 -0.008830  0.019963 -0.052564  0.283139  0.009457 -0.006094  0.002038
 0.003866 -0.011801  0.026547 -0.065390  0.215709  0.111334 -0.044861  0.014597
 0.002376 -0.007254  0.016305 -0.039468  0.106973  0.220817 -0.073183  0.023434
 0.000297 -0.000910  0.002047 -0.004817  0.007692  0.286206 -0.058021  0.017506
-0.001171  0.003573 -0.008027  0.019401 -0.052013  0.282347  0.010768 -0.004879
-0.001503  0.004591 -0.010324  0.024884 -0.062652  0.210630  0.120435 -0.036061
-0.000849  0.002598 -0.005862  0.014184 -0.035371  0.098994  0.235335 -0.059030
-0.000034  0.000108 -0.000264  0.000714 -0.001950  0.001921  0.296928 -0.047423
 0.000449 -0.001371  0.003084 -0.007431  0.018381 -0.050140  0.279179  0.007850
 0.000435 -0.001336  0.003042 -0.007467  0.018940 -0.050775  0.188558  0.098603
 0.000092 -0.000291  0.000710 -0.001924  0.005600 -0.017606  0.064040  0.199380
-0.000085  0.000257 -0.000572  0.001352 -0.003221  0.007714 -0.022894  0.267449
-0.000007  0.000034 -0.000138  0.000572 -0.002378  0.009892 -0.041146  0.283171
 0.000000  0.000000  0.000000 -0.000000 -0.000000 -0.000000  0.000000  0.250000
```

By comparing this new matrix with the exact solution on page 32 again, one can also see that the simplified bias-vector-free network converges towards the exact solution, this time with almost all matrix entries being identical up to the 6^{th} digit after the decimal point.

4.3 Retrieving Matrix M_2 Most Directly

The above solution is still not optimal. Both layers have trainable weights (hence we get two weight matrices `M1` and `M2`). By reducing the first layer to a mere `InputLayer` with no weights at all, we finally get the most direct representation:

$$\vec{q} = \underline{\underline{W_2}}\ \vec{Q} \tag{43}$$

The following code snippet shows the required modification to our program (compare with previous source code on page 40).

```
115 # ***********************************
    # Create and train Model
    # ***********************************
    print()
    print("**************")
120 print("Train model")
    print("**************")
    model=models.Sequential()
    model.add(layers.InputLayer(input_shape=(8,)))

125 model.add(layers.Dense(qcount, use_bias=False))
    model.summary()
    model.compile(optimizer=myoptimizer, loss='mse', metrics=['mae'])
    history=model.fit(train_data,train_targets,
                               epochs=maxepochs, batch_size=bSize,
130                            callbacks=callbacks_list,
                               validation_data=(val_data,val_targets))

    # *********************************************
    # Print effective Matrix and vector
135 # *********************************************
    W2=np.transpose(model.get_weights()[0]) # weight matrix input layer
    printMatrix("MATRIX W2",W2)
```

In line 123 the first layer is now a `InputLayer` with no weights. This is the output generated by this version of the program:

```
***************
PROGRAM STARTED
***************
Matrix2_8_4.txt already exists.
Load matrix file
Computing TrainSet 0 - 999
Computing TrainSet 100000 - 199999
Computing TrainSet 200000 - 299999
Computing TrainSet 300000 - 399999
Computing TrainSet 400000 - 499999
Computing TrainSet 500000 - 599999
Computing TrainSet 600000 - 699999
Computing TrainSet 700000 - 799999
Computing TrainSet 800000 - 899999
Computing TrainSet 900000 - 999999
Computing TrainSet 1000000 - 1099999
Computing TrainSet 1100000 - 1199999
Computing TrainSet 1200000 - 1299999
Computing TrainSet 1300000 - 1399999
Computing TrainSet 1400000 - 1499999
Computing TrainSet 1500000 - 1599999
Computing TrainSet 1600000 - 1699999
Computing TrainSet 1700000 - 1799999
```

```
**************
Train model
**************
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_10 (Dense)             (None, 32)                256
=================================================================
Total params: 256
Trainable params: 256
Non-trainable params: 0
_________________________________________________________________
Train on 1500000 samples, validate on 300000 samples
Epoch 1/10000
1500000/1500000 [==============================] - 3s 2us/step - loss: 0.1451 - mean_absolute_error: 0.2884 -
                                                   val_loss: 0.1347 - val_mean_absolute_error: 0.2757
...
Epoch 719/10000
1500000/1500000 [==============================] - 1s 1us/step - loss: 4.0106e-15 - mean_absolute_error: 3.3280e-08 -
                                                   val_loss: 4.1021e-15 - val_mean_absolute_error: 3.4008e-08

***************
MATRIX W2
***************
 0.250000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
 0.283171 -0.041146  0.009892 -0.002378  0.000572 -0.000138  0.000034 -0.000007
 0.267449 -0.022894  0.007714 -0.003222  0.001352 -0.000572  0.000257 -0.000084
 0.199380  0.064040 -0.017606  0.005600 -0.001924  0.000710 -0.000291  0.000092
 0.098603  0.188558 -0.050775  0.018940 -0.007467  0.003042 -0.001336  0.000435
 0.007850  0.279179 -0.050140  0.018380 -0.007431  0.003084 -0.001370  0.000449
-0.047423  0.296928  0.001921 -0.001950  0.000714 -0.000264  0.000109 -0.000034
-0.059030  0.235335  0.098994 -0.035371  0.014184 -0.005862  0.002598 -0.000849
-0.036061  0.120436  0.210630 -0.062652  0.024884 -0.010324  0.004591 -0.001503
-0.004879  0.010768  0.282347 -0.052013  0.019401 -0.008027  0.003573 -0.001171
 0.017506 -0.058021  0.286206  0.007692 -0.004817  0.002047 -0.000909  0.000297
 0.023434 -0.073183  0.220817  0.106973 -0.039468  0.016305 -0.007254  0.002376
 0.014597 -0.044861  0.111334  0.215708 -0.065390  0.026547 -0.011801  0.003866
 0.002038 -0.006094  0.009457  0.283138 -0.052564  0.019963 -0.008830  0.002892
-0.007094  0.021788 -0.053614  0.283772  0.008949 -0.005478  0.002496 -0.000819
-0.009541  0.029167 -0.067177  0.217381  0.109005 -0.041032  0.018136 -0.005939
-0.005939  0.018136 -0.041032  0.109005  0.217381 -0.067177  0.029167 -0.009541
-0.000819  0.002496 -0.005478  0.008949  0.283772 -0.053614  0.021788 -0.007094
 0.002892 -0.008830  0.019963 -0.052564  0.283139  0.009457 -0.006094  0.002038
 0.003866 -0.011801  0.026547 -0.065390  0.215708  0.111334 -0.044861  0.014597
 0.002376 -0.007254  0.016305 -0.039468  0.106973  0.220817 -0.073183  0.023434
 0.000297 -0.000909  0.002047 -0.004817  0.007692  0.286206 -0.058021  0.017506
-0.001171  0.003573 -0.008027  0.019401 -0.052013  0.282347  0.010768 -0.004879
-0.001503  0.004591 -0.010324  0.024884 -0.062652  0.210630  0.120436 -0.036061
-0.000849  0.002598 -0.005862  0.014184 -0.035371  0.098994  0.235335 -0.059030
-0.000034  0.000109 -0.000264  0.000714 -0.001950  0.001921  0.296928 -0.047423
 0.000449 -0.001371  0.003084 -0.007431  0.018380 -0.050140  0.279179  0.007850
 0.000435 -0.001336  0.003042 -0.007467  0.018940 -0.050775  0.188558  0.098603
 0.000092 -0.000291  0.000710 -0.001924  0.005600 -0.017606  0.064040  0.199380
-0.000084  0.000257 -0.000572  0.001352 -0.003221  0.007714 -0.022894  0.267448
-0.000007  0.000034 -0.000138  0.000572 -0.002378  0.009892 -0.041146  0.283171
-0.000000 -0.000000 -0.000000 -0.000000  0.000000 -0.000000 -0.000000  0.250000
```

This directly calculated matrix is identical to the exact solution with a precision of even more than the 6 printed digits after the decimal point.

4.4 Simple Convolutional Networks

4.4.1 Theory

As we have derived the exact solution for the charge-distribution problem in chapter 3.2, one could question whether it makes sense to deal with the concept of neural networks at all. If we have to deal with a reasonably small number of charges in the given one-dimensional setting, the answer is definitely no. The exact solution is easily and quickly calculated and free from numerical deviations.

However, in cases where the number of original charges reaches 1000 or more, calculating matrix $\underline{\underline{M_2}}$ becomes more and more time consuming (because of the required matrix inversion). What's more, most of the elements in these matrices turn out to have extremely small values. This is due to the following fact:

Let Q_j be the single original charge in the j-th Wigner-Seitz cell, and let n be the number of (smaller) sub-charges to be created for each original charge Q_j. Then the original charge Q_j will be replaced by charges $q_\alpha \dots q_\omega$, with $\alpha = (Q-1)n + 1$ and $\omega = nj$. Although in principle *all* charges $Q_1 \dots Q_{j_{max}}$ influence the values of $q_\alpha \dots q_\omega$, the actual influence of a specific charge Q_k on the values of $q_\alpha \dots q_\omega$ becomes smaller and smaller the farther away Q_k is from Q_j.

Therefore, for calculating $q_\alpha \dots q_\omega$ in good approximation, we can safely establish a method only considering a certain number of (left and right) neighboring charges of Q_j. It turns out this is just what a so-called (one-dimensional) *convolutional network* does in the domain of Deep Learning.

Figure 6 illustrates how this works in practice: A convolutional network usable for our purpose should have an odd number of input neurons (e.g. 5 neurons, as in Fig. 6). It "scans" the line of original charges $Q_1 \cdots Q_{j_{max}}$ one by one, and is always only calculating the refinement charges $q_\alpha \dots q_\omega$ for a single input charge Q_j at a time. In one step, the "center" input neuron gets the value of the currently considered charge Q_j as input (in Fig. 6 this is charge Q_6). The other input neurons receive values of the left and right neighboring charges. The total number of input neurons is called *kernel size.* The convolutional network depicted in Fig. 6 therefore has kernel size 5. The output layer delivers values for the refined charges $q_\alpha \dots q_\omega$ (in our example four output charges q_a, q_b, q_c, and q_d).

By scanning the whole list of original charges $Q_1 \dots Q_{j_{max}}$ one by one, e.g. from left to right, this relatively small network produces a list of almost all refined charges q_i. It's "almost all refined charges", because a network with kernel size s needs $\frac{s-1}{2}$ charges to the left and to the right of the currently considered charge Q_j as input. So, for example, a network with kernel size 5 (as in Fig. 6) can process neither the two leftmost charges Q_1 and Q_2, nor the two rightmost charges $Q_{j_{max}-1}$ and $Q_{j_{max}}$.

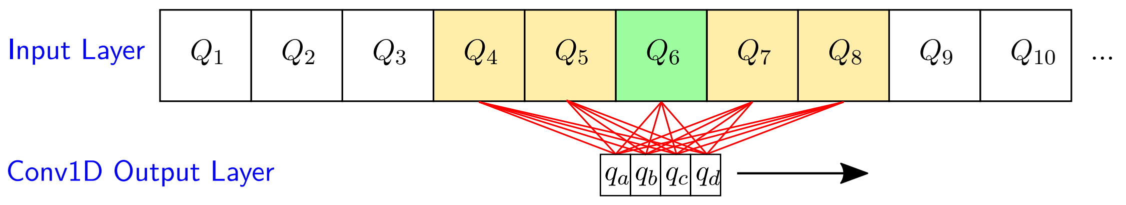

Figure 6: A one-dimensional convolutional network scanning an arbitrary long line of (coarse) charges Q_j from left to right. In every step it predicts the refined charges (here: $q_a \dots q_d$) which replace the single original charge Q_j in the center of the input layer (here it is currently processing charge Q_6). The depicted convolutional network uses 2 charges to the left and 2 charges to the right of the current charge, and hence has 5 input neurons (kernel size 5).

4.4.2 First Implementation in Python

In all previous programs we have implemented a fairly simple training strategy: The neural networks were trained against the `mse` loss function with a fixed learning rate, until there was no improvement for a certain number of epochs. However, we found that this training strategy does not work reliably with more refined network architectures (as we will present in the following chapters).

Firstly, when increasing the number of neurons, at first all network architectures improved as expected, up to a `mae` value in a magnitude between 10^{-4} and 10^{-5}. However, after a certain threshold of trainable parameters, some network architectures did not further improve beyond this `mae` magnitude. Worse, they sometimes even yielded an inferior result compared to the simpler versions.

It turns out that this problem can be overcome by a custom loss function that still returns comparably large absolute values for already well-trained networks where `mae` values are already small. Therefore, we have implemented a custom loss function (see lines 131-135 in the following source code listing). This custom loss function is simply the sum over all squared errors, multiplied with a factor of $\frac{10^{11}}{batchsize}$.

Secondly, the original, simple training method had some issues in the "end-game". With some networks, when the optimum was already reached, the optimizer (in search for a better optimum) did not stabilize the result. Instead, the loss function values sometimes began to fluctuate by one order of magnitude. So, for example, an end-result of 10^{-6} for the `mae` value could actually mean anything between 10^{-6} and 10^{-7}.

We solved this problem by employing the `ReduceLROnPlateau` callback. In our implementation, the learning rate is reduced with a factor of 0.2 if there is no improvement after 5 epochs (see lines 146-148 in the following source code).

As a further measure to mitigate this problem, the model is automatically saved after each epoch if it is better than the best-so-far model. This is realized by use of the `ModelCheckpoint` callback (see lines 142-144 in the following source code). After the training phase, the best model is reloaded and evaluated.

The complete source code is printed on the following three pages. The program trains a one-dimensional convolutional network for an initial charge distribution of 32 charges (line 14), to be split in 4 charges each (line 15). The kernel size (i.e. the total number of neighboring charges taken into account) is defined in line 18. The number of epochs after which the training should stop if there is no improvement is set to 20 (line 23). Total training set size is 1,000,000, with a validation set of size 200,000 and a test set size of 150,000 (lines 24-26). Batch size is chosen to be 100,000 (line 27), and again we use the `Adam` optimizer (line 28).

In lines 30-53, a function is implemented that plots the original charges, the actual target charge values, and the refined charges as produced by the trained network. Starting from line 55, a matching $\underline{\underline{M_2}}$ matrix file is generated (if not already available) by calling the `MatrixGen`-program from chapter 3.3.5, and then matrix $\underline{\underline{M_2}}$ is loaded for further use.

Eventually, in lines 91-128, data sets for training, validating and testing are generated by utilizing the $\underline{\underline{M_2}}$ matrix and the `chrg_generator` helper function (lines 80-89).

In lines 131-149, the already explained custom loss function and callbacks are defined (including the `EarlyStopping` callback, which stops the training after `mypat` epochs without improvement).

Starting from line 150, a one-dimensional convolutional network with just one trainable layer (no bias weights) is created and trained. Beginning in line 168, the best model is re-loaded and `mae` values are printed. Eventually, in lines 180-190, the network's performance is visualized by plotting the network's output of one record in the test data set next to the actual target values.

```python
"""
CHARGE REFINEMENT BY CONVOLUTIONAL NETWORK
V 1.0, (C) 2019 HELMUT HOERNER
File: CR_conf22savbest.py
"""
import os
import numpy as np
import matplotlib.pyplot as plt
import keras.backend as KBE
from keras import models
from keras import layers
from keras import callbacks

QCount=32
SplitFactor=4
FileName = 'Matrix2_'+str(QCount)+'_'+str(SplitFactor)+'.txt'

kernel=25
padding=int((kernel-1)/2) # cells left and right to be left out
np.random.seed(0) # make pseudo random numbers reproducible

maxepochs=5000 # max number of epochs
mypat=20 # stop after this no of epochs if no improvement
trainSetSize=1000000 # training data
valSetSize  = 200000 # validation data
testSetSize = 150000 #test data
bSize=100000 # batch Size
myoptimizer='Adam' # optimizer

# **********************************
# Charge Plotting Function
# **********************************
def pltChrge(Q, q, q_pred, lines=False, title='',legend=False):
    plt.figure(figsize=(16, 8), dpi=150)
    cellBorders=[i*SplitFactor-0.5 for i in range(0, Qcount+1)]
    # x positions of the full and split charges (not yet refined)
    Qx=[i*SplitFactor+float(SplitFactor)/2.-0.5 for i in range(0, Qcount)]
    qx=[i for i in range(0, Qcount*SplitFactor)]
    for xc in cellBorders:
        plt.axvline(x=xc, color='gray')

    plt.plot(Qx,Q,'ro', markersize=12, label='full charges')
    plt.plot(qx,q*SplitFactor,'bo',label='split charges*'+str(SplitFactor))
    plt.plot(qx,q_pred*SplitFactor,'go',label='predition*'+str(SplitFactor))

    if lines:
        plt.plot(qx,q*SplitFactor,'b')
        plt.plot(qx,q_pred*SplitFactor,'g')
    if legend:
        plt.legend()
    if title!='':
        plt.title(title)
    return()

# ******************************************
# Generate and load matrix M2
# ******************************************
if os.path.isfile(FileName):
    print(FileName,"already exists.")
else:
    print('Generating matrix file',FileName)
    os.system('MatrixGen '+str(QCount)+' '+str(SplitFactor)+' 2')

print('Load matrix file')
f=open(FileName,encoding="utf-8")
MatrixData=f.read()
f.close()
MatrixLines=MatrixData.split('\n')
Qcount = len(MatrixLines[0].split(','))
qcount = len(MatrixLines)
SplitFactor = int(qcount/Qcount)

# parsing matrix data
M2 = np.zeros((qcount, Qcount))
for i, line in enumerate(MatrixLines):
    sline=line.split(',')
    values = [float(x) for x in sline]
    M2[i,:]=values
```

```
# **********************************
# Charge Generator
# **********************************
def chrg_generator(n):
    """ Creates test set with n entries """
    for i in range(n):
        # Creates random full charges
        Q=np.random.rand(Qcount)
        q=M2@Q
        yield([i-1, q, Q])

# *************************************
# Generate data
# *************************************
totSize=trainSetSize+valSetSize+testSetSize
data=np.zeros((totSize,Qcount))
targets=np.zeros((totSize,qcount))
targets_c=np.zeros((totSize,qcount-2*padding*SplitFactor))
offset=0
for i, q, Q in chrg_generator(trainSetSize+valSetSize+testSetSize):
    if i%100000==0:
        print("Computing TrainSet",i,"-",i+99999)
    data[i]=Q
    targets[i]=q
    targets_c[i]=q[padding*SplitFactor:-padding*SplitFactor]
#re-shape input data for conv network
exp_data=np.expand_dims(data, axis=2)

# Separate Train Data
train_data=data[:trainSetSize]
exp_train_data=exp_data[:trainSetSize]
train_targets=targets[:trainSetSize]
train_targets_c=targets_c[:trainSetSize]

# Separate Validation Data
val_data=data[trainSetSize:trainSetSize+valSetSize]
exp_val_data=exp_data[trainSetSize:trainSetSize+valSetSize]
val_targets=targets[trainSetSize:trainSetSize+valSetSize]
val_targets_c=targets_c[trainSetSize:trainSetSize+valSetSize]

# Separate Test Data
test_data=data[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
exp_test_data=exp_data[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
test_targets=targets[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
test_targets_c=targets_c[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]


# ***********************************
# Custom Loss Function
# ***********************************
def custom_loss(yTrue,yPred):
    return KBE.sum(KBE.square(yTrue - yPred))*(1E11/bSize)

# **********************************
# callback checkpoints
# **********************************
bestmodelfile='BestModel_C_'+str(kernel)+'_'
bestmodelfile=bestmodelfile+str(bSize)+'_'+str(mypat)+'.hdf5'
checkpoint = callbacks.ModelCheckpoint(bestmodelfile, monitor='val_loss',
                                       verbose=1, save_best_only=True,
                                       mode='min')
earlystopping=callbacks.EarlyStopping(monitor='loss', patience=mypat,)
reduce_lr_loss = callbacks.ReduceLROnPlateau(monitor='loss', factor=0.2,
                                             patience=5, verbose=1,
                                             epsilon=1e-4, mode='min')
callbacks_list = [checkpoint, earlystopping, reduce_lr_loss]
```

```
# **********************************
# Create and train Conv1D Model
# **********************************
print("**************")
print("Train Conv1D model, excluding border 2 *",padding)

model=models.Sequential()
model.add(layers.Conv1D(filters=SplitFactor, use_bias=False,
                        kernel_size=kernel,
                        input_shape=(Qcount,1)))
model.add(layers.Flatten())
model.summary()
model.compile(optimizer=myoptimizer, loss=custom_loss, metrics=['mae'])
history=model.fit(exp_train_data,train_targets_c,
                      epochs=maxepochs, batch_size=bSize,
                      callbacks=callbacks_list,
                      validation_data=(exp_val_data,val_targets_c))

# ***************************
# Load and test best model
# ***************************
model.load_weights(bestmodelfile)
train_score=model.evaluate(exp_train_data, train_targets_c)
val_score=model.evaluate(exp_val_data, val_targets_c)
test_score=model.evaluate(exp_test_data, test_targets_c)
print ("Train mae ", train_score[1])
print ("Validation mae ", val_score[1])
print ("Test mae ", test_score[1])


# ***************************************************
# Make prediction on first test set record
# ***************************************************
prediction=np.zeros(qcount) # empty array

# predict core charges using Conv1D model
prediction[padding*SplitFactor:-padding*SplitFactor]= \
  model.predict(exp_test_data[0:1])[0]

pltChrge(test_data[0], test_targets[0],
         prediction, True, '',True)
```

4.4.3 Results

Figure 8 visualizes how the output of convolutional networks of kernel sizes 3, 5, 7, and 9 compares to the actual target output. As expected, a network with kernel size 3 does not do a very satisfactory job, as it always considers just one charge on the left side and one charge on the right of the currently processed charge Q_j. With increasing kernel size, accuracy improves visibly (see Fig. 8).

As can be seen in Table 1 and Figure 7, accuracy is well-scaleable. Whereas the simplest dense network, as presented in chapter 4.3, still needs to train 256 parameters, the convolutional network reaches e.g. a respectable `mae` value of $9.95 \cdot 10^{-5}$ with only 52 parameters. With 100 parameters (still less that 40% of the dense network's parameters), `mae` improves to $5.09 \cdot 10^{-7}$.

Kernel Size	**Parameters**	**mae**
3	12	0.008837651
5	20	0.003594638
7	28	0.001465990
9	36	0.000597947
11	44	0.000243941
13	52	0.000099474
15	60	0.000040561
17	68	0.000016541
19	76	0.000006742
21	84	0.000002746
23	92	0.000001136
25	100	0.000000509

Table 1: Mean absolute error over a test set of 150,000 records produced by convolutional networks with various kernel sizes (each trained with 1,000,000 training records and 200,000 validation records). The task was to create 4 smoothly distributed sub-charges each, for a total of 32 original (coarse) charges. The middle column shows the number of trained parameters (no bias weights).

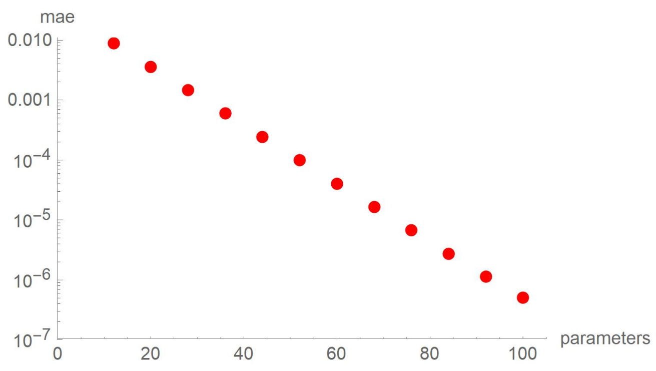

Figure 7: Trade-off between number of trained parameters (due to different kernel sizes) and mean average error in a simple convolutional network.

4.5 Improved Implementation in Python with Handling of Boundary Charges

Unfortunately, as recognizable in Fig. 8, with increased kernel size not only the accuracy, but also the number of charges not covered by the method on the left and on the right boundary, increases. To overcome this boundary problem, we have implemented an enhanced version of the software. The source code of this version is printed in Appendix 6.4.

These are the main differences: In lines 116, 124 and 135-136, additional record sets, containing training-, validation- and test-data for just the left boundary charges are created. Then, starting from line 141, a dense network is trained just for the left boundary charges. Eventually, in line 183 this model is used to predict the left boundary charges. Because of the symmetry of the problem, this model can also be used to predict the right boundary charges (see lines 190-191). To do so, the right boundary charges are reversed before being fed into the model for the left boundary. Then, the output of the model is reversed once more.

4.5.1 Results

Figure 9 shows the output of the improved software, again the convolutional network part employing kernel sizes 3, 5, 7, and 9. The border charges fit seamlessly to the charges of the convolutional model .

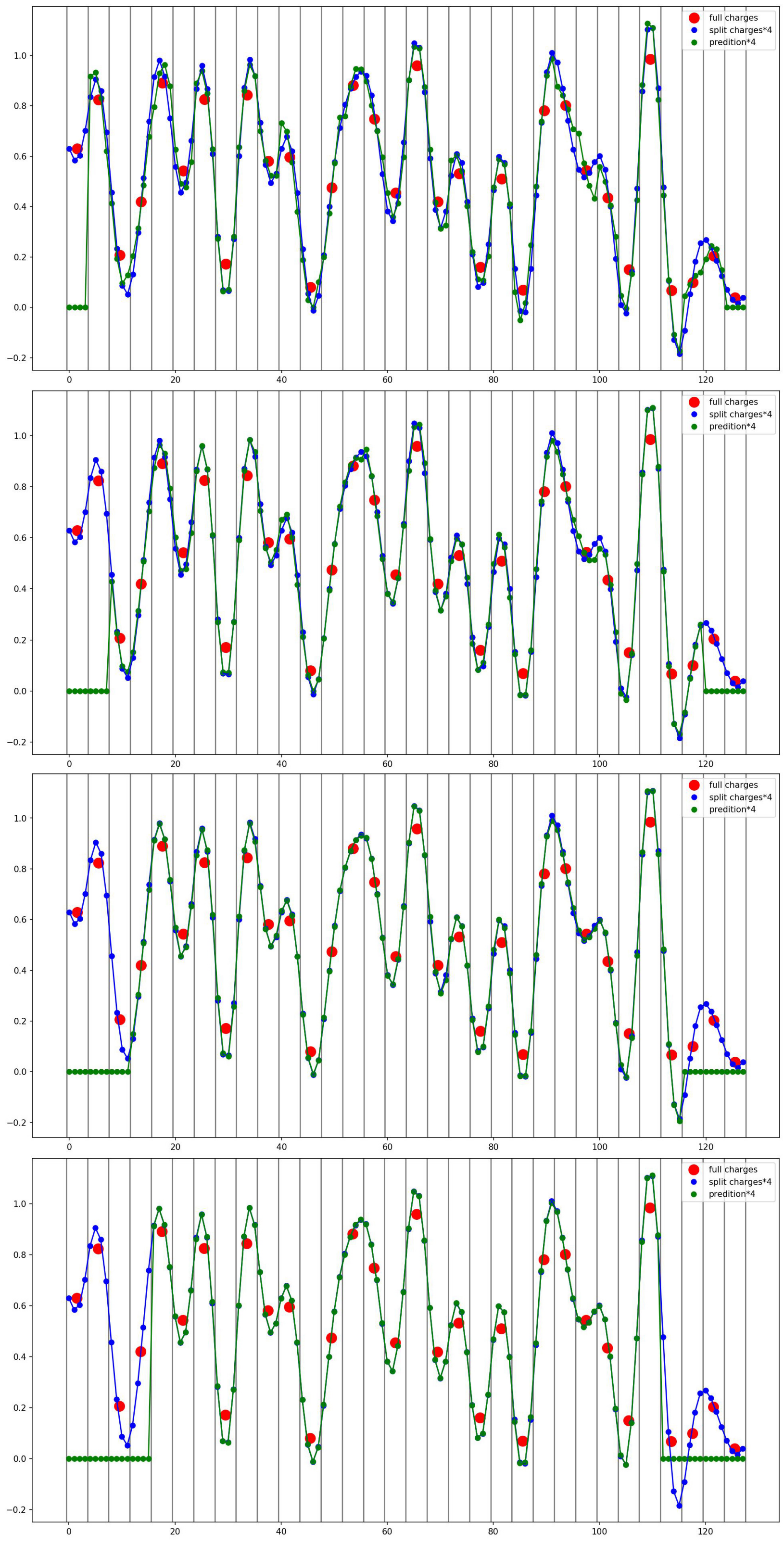

Figure 8: Performance of convolutional networks with kernel sizes 3, 5, 7, and 9.

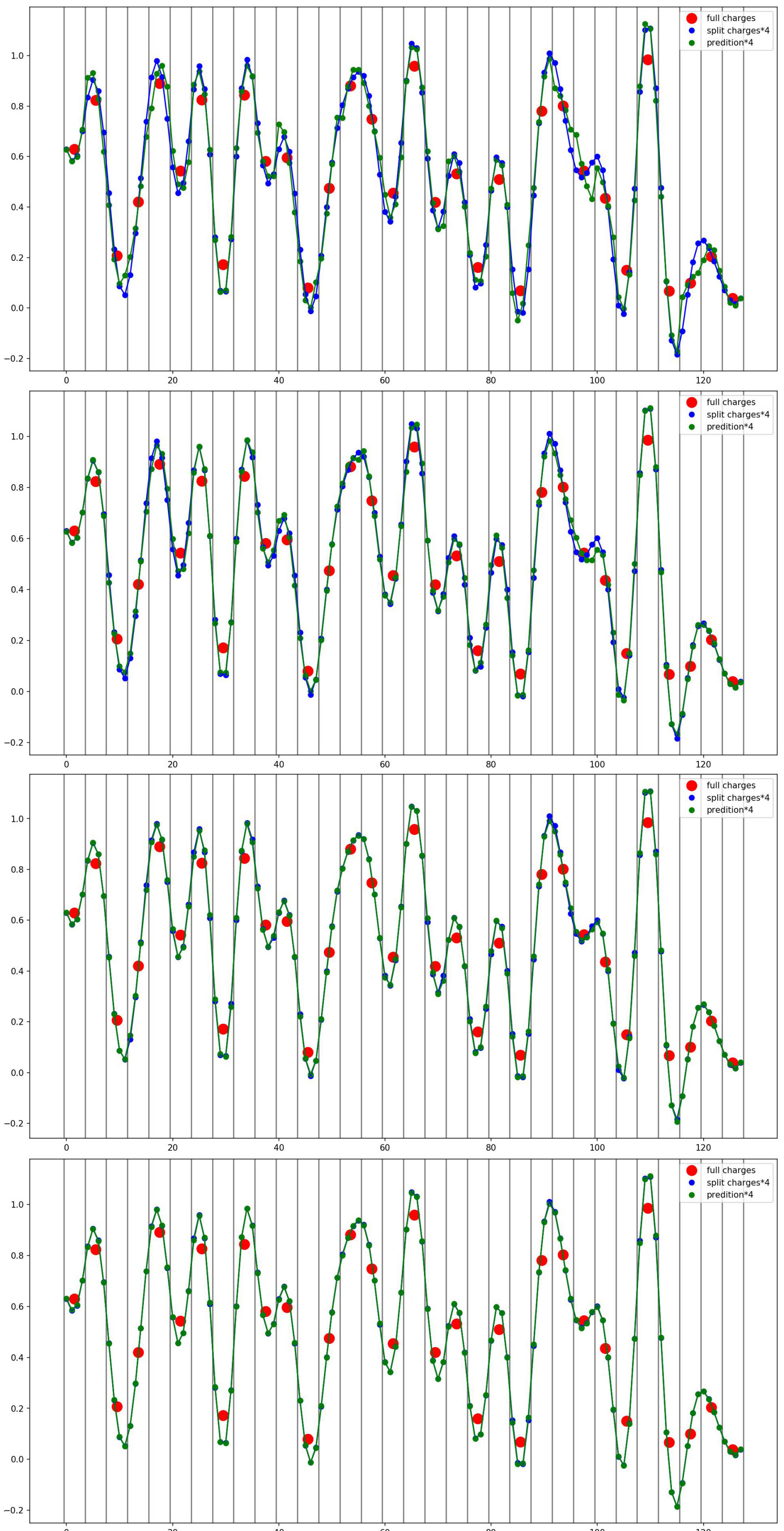

Figure 9: Improved software with kernel sizes 3, 5, 7, 9 and boundary charge handling.

4.6 Better Results with Less Neurons

4.6.1 A Refined Architecture - The Basic Idea

The smaller a network gets (i.e. the less parameters it has to process), the faster it becomes. Therefore, the simple convolutional network presented in chapter 4.4 was already a significant improvement over the original dense network in chapter 4.3.

It is logical to ask whether there could be a network architecture that can produce the same or better results with even smaller networks. In this chapter we present such an architecture.

Figure 10 demonstrates the principal idea. A certain number of charges left and right of the currently processed charge is handled directly by a convolutional network layer as before. In Figure 10, the currently processed charge is charge Q_6 (depicted in green), and the two adjacent charges Q_5 and Q_7 (depicted in orange) are directly processed by the `Conv1D Output Layer`. Of course, in general more than these two neighboring charges could be processed directly. We shall call the total number of charges directly processed in this way the *core kernel size*. The network in Figure 10 has a *core kernel size* of 3.

However, information about more distant neighbor charges on the left and right side is not fed into the `Conv1D Output Layer` directly. Instead, left and right distant charge values are cumulated into one single value each by the `left Conv1D Layer` and the `right Conv1D Layer`. Only these cumulated values are then fed into the `Conv1D Output Layer`. In the example presented in Figure 10, the distant charges Q_3 and Q_4 on the left side, and distant charges Q_8 and Q_9 on the right side are cumulated into L_{34} and R_{89} (in yellow). We shall call the number of distant charges cumulated in this way on either side the *border kernel size*. The network in Figure 10 has a *border kernel size* of 2.

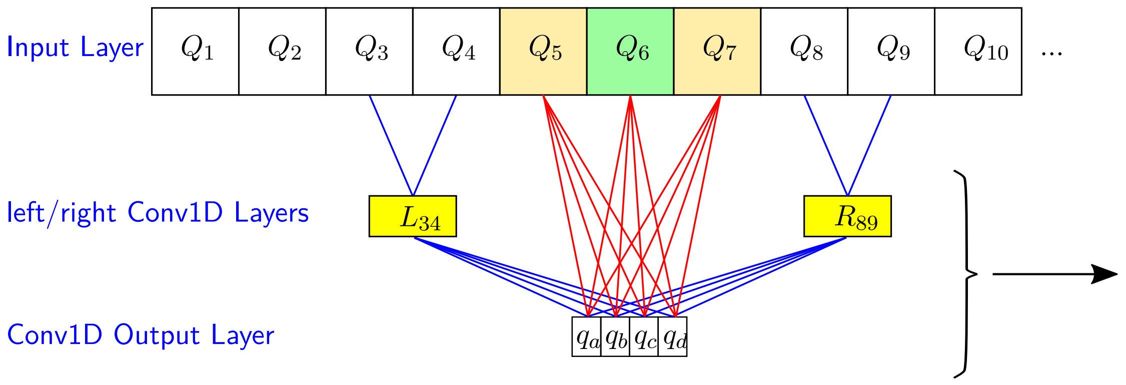

Figure 10: An improved architecture: Only a certain number of charges left and right of the center charge is handled directly by the output layer. Remote charges are condensed on the left and right side separately before being further processed.

4.6.2 The Actual Implementation

Figure 11 shows how the above presented architecture was actually implemented: Two convolutional layers `avgL` and `avgR` calculate the left and right side cumulative values. Then, a stack of layers (`laystack`) is created by copying and cropping layers `avgL`, `lay_inp` and `avgR`, so that all values to be fed into the output layer in each step are occupying the same index position.

Finally, this stack is merged and flattened out into layer `merge_flat`, so that the final convolutional `core` layer can process all relevant data in each step. The `core` layer moves with according stride. In our example, it moves 5 neurons in every step.

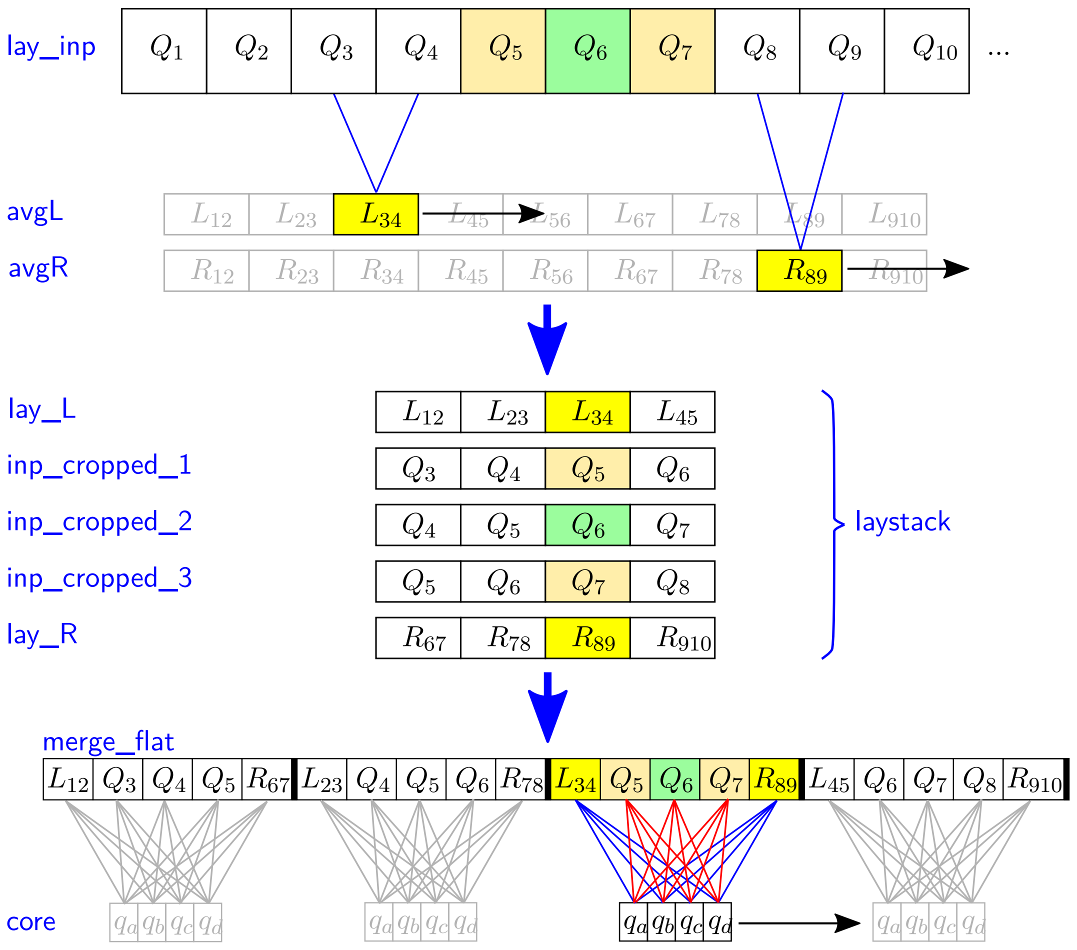
lay_inp
Q_1 Q_2 Q_3 Q_4 Q_5 Q_6 Q_7 Q_8 Q_9 Q_{10} …
avgL
L_{12} L_{23} L_{34} L_{45} L_{56} L_{67} L_{78} L_{89} L_{910}
avgR
R_{12} R_{23} R_{34} R_{45} R_{56} R_{67} R_{78} R_{89} R_{910}
lay_L
L_{12} L_{23} L_{34} L_{45}
inp_cropped_1
Q_3 Q_4 Q_5 Q_6
inp_cropped_2
Q_4 Q_5 Q_6 Q_7
inp_cropped_3
Q_5 Q_6 Q_7 Q_8
lay_R
R_{67} R_{78} R_{89} R_{910}
laystack
merge_flat
L_{12} Q_3 Q_4 Q_5 R_{67} L_{23} Q_4 Q_5 Q_6 R_{78} L_{34} Q_5 Q_6 Q_7 R_{89} L_{45} Q_6 Q_7 Q_8 R_{910}
core
q_a q_b q_c q_d

Figure 11: The actual implementation of the architecture presented in Figure 10.

The complete source code is printed on the following three pages. It is almost identical to the previous source code, except (of course) for the different network architecture, implemented in code lines 152-192.

All other minor differences originate from the fact that we now have two parameters `kernel_c` and `kernel_b`, representing the *core kernel size* and the *border kernel size*.

```
"""
CHARGE REFINEMENT BY IMPROVED CONVOLUTIONAL NETWORK
V 1.0, (C) 2019 HELMUT HOERNER
File: CR_conf20Bsavbest.py
"""
import os
import numpy as np
import matplotlib.pyplot as plt
import keras.backend as KBE
from keras import models
from keras import layers
from keras import callbacks

kernel_c=3
kernel_b=2
QCount=32
SplitFactor=4
FileName = 'Matrix2_'+str(QCount)+'_'+str(SplitFactor)+'.txt'

maxepochs=5000 # max number of epochs
mypat=20 # stop after this no of epochs if no improvement
trainSetSize=1000000 # training data
valSetSize  = 200000 # validation data
testSetSize = 150000 #test data
bSize=100000 # batch Size
myoptimizer='Adam' # optimizer

# **********************************
# Charge Plotting Function
# **********************************
def pltChrge(Q, q, q_pred, lines=False, title='',legend=False):
    plt.figure(figsize=(16, 8), dpi=150)
    cellBorders=[i*SplitFactor-0.5 for i in range(0, Qcount+1)]
    # x positions of the full and split charges (not yet refined)
    Qx=[i*SplitFactor+float(SplitFactor)/2.-0.5 for i in range(0, Qcount)]
    qx=[i for i in range(0, Qcount*SplitFactor)]
    for xc in cellBorders:
        plt.axvline(x=xc, color='gray')

    plt.plot(Qx,Q,'ro', markersize=12, label='full charges')
    plt.plot(qx,q*SplitFactor,'bo',label='split charges*'+str(SplitFactor))
    plt.plot(qx,q_pred*SplitFactor,'go',label='predition*'+str(SplitFactor))

    if lines:
        plt.plot(qx,q*SplitFactor,'b')
        plt.plot(qx,q_pred*SplitFactor,'g')
    if legend:
        plt.legend()
    if title!='':
        plt.title(title)
    return()

# ********************************************
# Generate and load matrix M2
# ********************************************
if os.path.isfile(FileName):
    print(FileName,"already exists.")
else:
    print('Generating matrix file',FileName)
    os.system('MatrixGen '+str(QCount)+' '+str(SplitFactor)+' 2')

print('Load matrix file')
f=open(FileName,encoding="utf-8")
MatrixData=f.read()
f.close()
MatrixLines=MatrixData.split('\n')
Qcount = len(MatrixLines[0].split(','))
qcount = len(MatrixLines)
SplitFactor = int(qcount/Qcount)

# parsing matrix data
M2 = np.zeros((qcount, Qcount))
for i, line in enumerate(MatrixLines):
    sline=line.split(',')
    values = [float(x) for x in sline]
    M2[i,:]=values
```

```
# **********************************
# Charge Generator
# **********************************
def chrg_generator(n):
    """ Creates test set with n entries """
    for i in range(n):
        # Creates random full charges
        Q=np.random.rand(Qcount)
        q=M2@Q
        yield([i-1, q, Q])


padding=kernel_b+int((kernel_c-1)/2)
np.random.seed(0) # make pseudo random numbers reproducible

# *************************************
# Generate data
# *************************************
totSize=trainSetSize+valSetSize+testSetSize
data=np.zeros((totSize,Qcount))
targets=np.zeros((totSize,qcount))
targets_c=np.zeros((totSize,qcount-2*padding*SplitFactor))
offset=0
for i, q, Q in chrg_generator(trainSetSize+valSetSize+testSetSize):
    if i%100000==0:
        print("Computing TrainSet",i,"-",i+99999)
    data[i]=Q
    targets[i]=q
    targets_c[i]=q[padding*SplitFactor:-padding*SplitFactor]
#re-shape input data for conv network
exp_data=np.expand_dims(data, axis=2)

# Separate Train Data
train_data=data[:trainSetSize]
exp_train_data=exp_data[:trainSetSize]
train_targets=targets[:trainSetSize]
train_targets_c=targets_c[:trainSetSize]

# Separate Validation Data
val_data=data[trainSetSize:trainSetSize+valSetSize]
exp_val_data=exp_data[trainSetSize:trainSetSize+valSetSize]
val_targets=targets[trainSetSize:trainSetSize+valSetSize]
val_targets_c=targets_c[trainSetSize:trainSetSize+valSetSize]

# Separate Test Data
test_data=data[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
exp_test_data=exp_data[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
test_targets=targets[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
test_targets_c=targets_c[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]

# **********************************
# Custom Loss Function
# **********************************
def custom_loss(yTrue,yPred):
    return KBE.sum(KBE.square(yTrue - yPred))*(1E11/bSize)

# **********************************
# callback checkpoints
# **********************************
bestmodelfile='Test_BestModel_A_'+str(kernel_b)+'_'+str(kernel_c)+'_'
bestmodelfile=bestmodelfile+str(bSize)+'_'+str(mypat)+'.hdf5'
checkpoint = callbacks.ModelCheckpoint(bestmodelfile, monitor='val_loss',
                                       verbose=1, save_best_only=True,
                                       mode='min')
earlystopping=callbacks.EarlyStopping(monitor='loss', patience=mypat,)
reduce_lr_loss = callbacks.ReduceLROnPlateau(monitor='loss', factor=0.2,
                                             patience=5, verbose=1,
                                             epsilon=1e-4, mode='min')
callbacks_list = [checkpoint, earlystopping, reduce_lr_loss]
```

```
# ****************************
# Create and train model
# ****************************
print("**************")
print("Train model, exluding border 2 *",padding)
print("Effective kernel size ",2*kernel_b+kernel_c)

lay_inp=layers.Input(shape=(Qcount,1), name="input")
lay_avgL=layers.Conv1D(filters=1, use_bias=False, kernel_size=kernel_b,
                    strides=1, name="avgL")(lay_inp)
lay_avgR=layers.Conv1D(filters=1, use_bias=False, kernel_size=kernel_b,
                    strides=1,
                    name="avgR")(lay_inp)

padding_border=int(2*padding-kernel_b+1)
laystack=[layers.Cropping1D(cropping=(0, padding_border),
                           name="lay_L")(lay_avgL)]
laystack=laystack+ \
        [layers.Cropping1D(cropping=(i, int(2*padding-i)))(lay_inp) \
        for i in range(kernel_b, kernel_b+kernel_c)]
laystack=laystack+[layers.Cropping1D(cropping=(padding_border, 0),
                                  name="lay_R")(lay_avgR)]

lay_merge=layers.concatenate(laystack, axis=-1, name="merge")

kernel=len(laystack)
flatlayersize=int((QCount-2*padding)*kernel)
lay_merge_flat=layers.Reshape((flatlayersize,1), name="merge_flat")(lay_merge)

lay_core=layers.Conv1D(filters=SplitFactor, use_bias=False,
                        kernel_size=kernel, strides=kernel,
                        name="core")(lay_merge_flat)

lay_outp=layers.Flatten(name="output")(lay_core)
model=models.Model(lay_inp, lay_outp)
model.summary()
model.compile(optimizer=myoptimizer, loss=custom_loss, metrics=['mae'])
history=model.fit(exp_train_data,train_targets_c,
                       epochs=maxepochs, batch_size=bSize,
                       callbacks=callbacks_list,
                       validation_data=(exp_val_data,val_targets_c))

# ****************************
# Load and test best model
# ****************************
model.load_weights(bestmodelfile)
train_score=model.evaluate(exp_train_data, train_targets_c)
val_score=model.evaluate(exp_val_data, val_targets_c)
test_score=model.evaluate(exp_test_data, test_targets_c)
print ("Train mae ", train_score[1])
print ("Validation mae ", val_score[1])
print ("Test mae ", test_score[1])

# **************************************************
# Make prediction on first test set record
# **************************************************
prediction=np.zeros(qcount) # empty array

# predict core charges using Conv1D model
prediction[padding*SplitFactor:-padding*SplitFactor]= \
  model.predict(exp_test_data[0:1])[0]

pltChrge(test_data[0], test_targets[0],
         prediction, True, '',True)
```

4.6.3 Results

Figure 12 and Table 2 show results achieved with networks of this improved architecture. A network with just 30 parameters (*core kernel size=3, border kernel size=5*) does now achieve the same `mae` value in the magnitude of 10^{-4} as before a simple convolutional network with 52 parameters. That's an improvement of 42%

Even better, a network with just 50 parameters (*core kernel size=5, border kernel size=11*) does now achieve a `mae` value of about $5 \cdot 10^{-7}$. The previous simple convolutional network required 100 parameters for that, which means that the current network provides a 50% improvement!

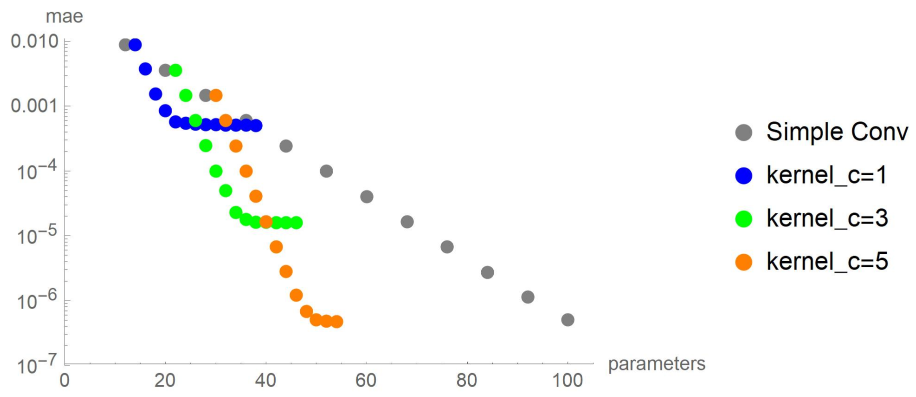

Figure 12: Networks with improved convolutional architecture compared to simple convolutional networks with different kernel sizes. For any improved network with a given *core kernel size* (`kernel_c`), the *border kernel size* has been varied between 1 and 13.

Core Kernel Size	**Border Kernel Size**	**Parameters**	**mae**
1	1	14	0.008837629
	2	16	0.003734775
	3	18	0.001531520
	4	20	0.000854907
	5	22	0.000572012
	6	24	0.000547836
	7	26	0.000523856
	8	28	0.000523109
	9	30	0.000517452
	10	32	0.000514900
	11	34	0.000511788
	12	36	0.000509279
	13	38	0.000506662
3	1	22	0.003594634
	2	24	0.001472544
	3	26	0.000599518
	4	28	0.000246431
	5	30	0.000100421
	6	32	0.000049600
	7	34	0.000023020
	8	36	0.000017994
	9	38	0.000016279
	10	40	0.000016203
	11	42	0.000016058
	12	44	0.000016041
	13	46	0.000015983
5	1	30	0.001465993
	2	32	0.000599119
	3	34	0.000244479
	4	36	0.000099735
	5	38	0.000040654
	6	40	0.000016613
	7	42	0.000006756
	8	44	0.000002815
	9	46	0.000001209
	10	48	0.000000680
	11	50	0.000000508
	12	52	0.000000486
	13	54	0.000000474

Table 2: Mean absolute error over a test set of 150,000 records produced by convolutional networks with improved architecture and various core and border kernel sizes (each trained with 1,000,000 training records and 200,000 validation records). The task was to create 4 smoothly distributed sub-charges each, for a total of 32 original (coarse) charges.

4.7 Further Slim Down the Neural Net

4.7.1 Exploiting Mirror Symmetry

Is the improved network presented in the previous chapter the end of the line when it comes to making the neural network more lightweight? The answer is: No, it isn't. There is a mirror symmetry waiting to be be exploited for a further slim-down.

In the previous implementation we had to use (and train) two different layers for handling the left and the right remote border charges separately (layers `avgL` and `avgR` in Figure 10). The reason is that layer `avgL` needs to give more weight to right-side charges, as these are closer to the currently processed core charge, whereas layer `avgR` needs to give - for the same reason - more weight to left-side charges.

But then, the problem is completely mirror-symmetric. If we just flipped the order of the input charges, we could use layer `avgL` to also process remote right charges (and vice versa)!

This idea leads to the further improved network architecture explained in the following sub-chapter, where there is only one convolutional layer for averaging the remote border charges on both sides, instead of hitherto two layers.

4.7.2 The Actual Implementation

Figure 13 shows how this idea can be converted into an actual software architecture. Firstly, the input layer `lay_inp` is mirrored into layer `lay_inpRev`, and both layers are then spliced together into layer `inp_dbl`. This layer now contains all input charges twice: First in original order, and then in reverse order.

Instead of previously two layers (`avgL` and `avgL`), now only one convolutional layer `avg` is averaging the border charges. The left half of this layer now contains all averaged left remote charges, and the right side all averaged right remote charges (the latter still in reverse order).

Therefore, the left part of layer `avg` is copied into layer `avgL`, and the right part into layer `avgRrev`, which is eventually back-reversed into a layer `avgR`.

The rest of the procedure is exactly the same as before: A stack of layers (`laystack`) is created by copying and cropping layers `avgL`, `lay_inp` and `avgR`, so that all values to be fed into the output layer in each step are occupying the same index position.

Finally, this stack is merged and flattened out into layer `merge_flat`, so that the final convolutional `core` layer can process all relevant data in each step. The `core` layer moves with according stride. In our example, it moves 5 neurons in every step.

The source code snippet on page 64 shows the actual Python implementation of the above explained software architecture. It replaces the according part of the previous program. The rest of the program remains unchanged.

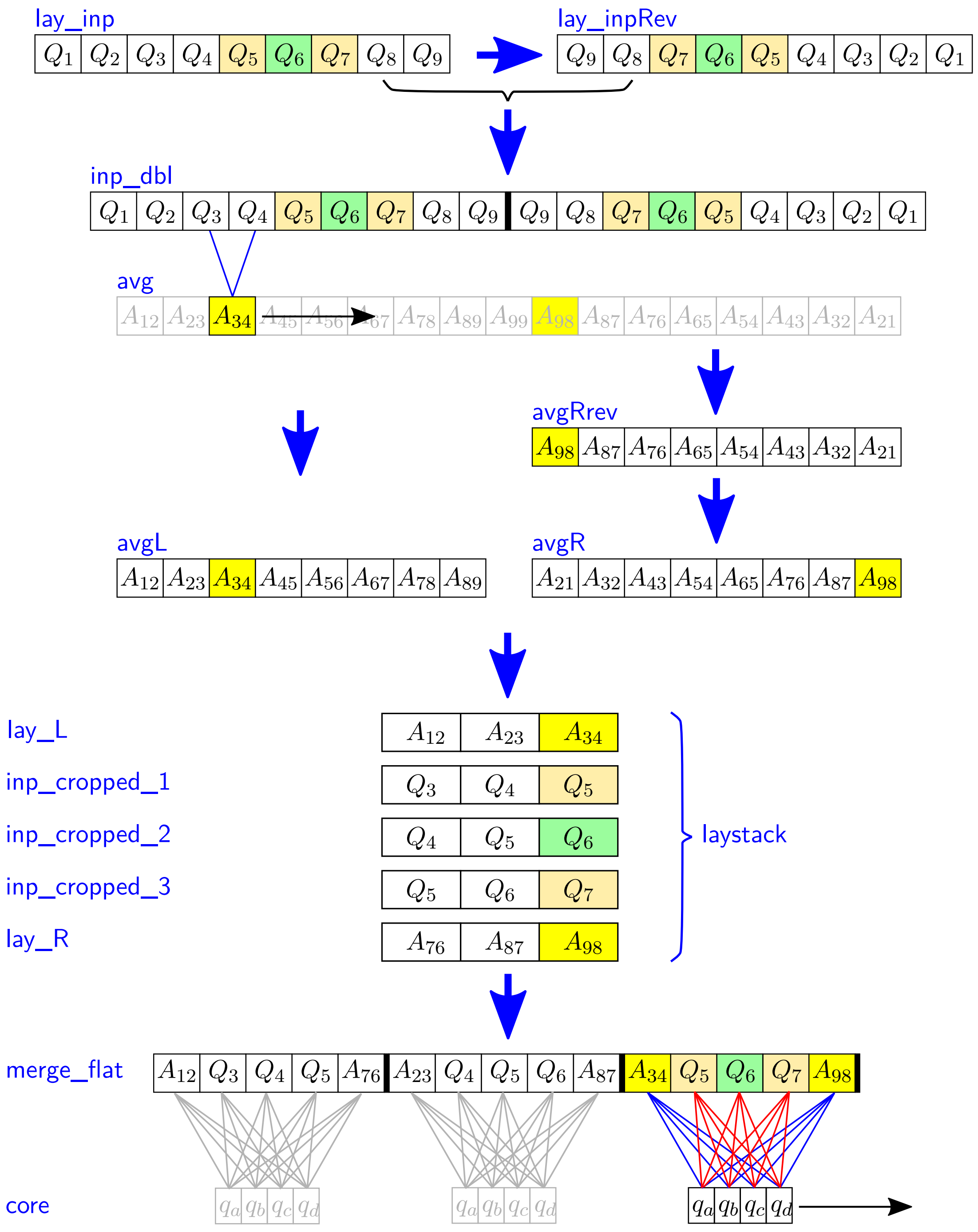
lay_inp
Q_1 Q_2 Q_3 Q_4 Q_5 Q_6 Q_7 Q_8 Q_9
lay_inpRev
Q_9 Q_8 Q_7 Q_6 Q_5 Q_4 Q_3 Q_2 Q_1
inp_dbl
avg
A_{12} A_{23} A_{34} A_{45} A_{56} A_{67} A_{78} A_{89} A_{99} A_{98} A_{87} A_{76} A_{65} A_{54} A_{43} A_{32} A_{21}
avgRrev
avgL
avgR
lay_L
inp_cropped_1
inp_cropped_2
inp_cropped_3
lay_R
laystack
merge_flat
core
q_a q_b q_c q_d

Figure 13: The improved convolutional network can be further slimmed-down by exploiting mirror symmetry.

```python
# ****************************
# Create and train Model
# ****************************
print("*************")
print("Train model, exluding border 2 *",padding)
print("Effective kernel size ",2*kernel_b+kernel_c)

lay_inp=layers.Input(shape=(Qcount,1), name="lay_inp")
lay_inpRev=layers.Lambda(lambda x: KBE.reverse(x,axes=1),
                          name="lay_inpRev")(lay_inp)
inp_dbl=layers.concatenate([lay_inp, lay_inpRev], axis=1, name="inp_dbl")

lay_avg=layers.Conv1D(filters=1, use_bias=False, kernel_size=kernel_b,
                   strides=1, name="avg")(inp_dbl)

lay_avg_size=int(lay_avg.shape[1])
lay_avgL_size=Qcount-kernel_b+1
crop_size=lay_avg_size-lay_avgL_size

lay_avgL=layers.Cropping1D(cropping=(0,crop_size), name="avgL")(lay_avg)
lay_avgRrev=layers.Cropping1D(cropping=(crop_size,0),
                               name="avgRrev")(lay_avg)
lay_avgR=layers.Lambda(lambda x: KBE.reverse(x,axes=1),
                          name="lay_avgR")(lay_avgRrev)

padding_border=int(2*padding-kernel_b+1)
laystack=[layers.Cropping1D(cropping=(0, padding_border),
                          name="lay_L")(lay_avgL)]
laystack=laystack+ \
        [layers.Cropping1D(cropping=(i, int(2*padding-i)))(lay_inp) \
        for i in range(kernel_b, kernel_b+kernel_c)]

laystack=laystack+[layers.Cropping1D(cropping=(padding_border, 0),
                                      name="lay_R")(lay_avgR)]

lay_merge=layers.concatenate(laystack, axis=-1, name="merge")

kernel=len(laystack)
flatlayersize=int((QCount-2*padding)*kernel)
lay_merge_flat=layers.Reshape((flatlayersize,1),
                               name="merge_flat")(lay_merge)

lay_core=layers.Conv1D(filters=SplitFactor, use_bias=False,
                       kernel_size=kernel, strides=kernel,
                       name="core")(lay_merge_flat)

lay_outp=layers.Flatten(name="output")(lay_core)

model=models.Model(lay_inp, lay_outp)
model.summary()
model.compile(optimizer=myoptimizer, loss=custom_loss, metrics=['mae'])
history=model.fit(exp_train_data,train_targets_c,
                      epochs=maxepochs, batch_size=bSize,
                      callbacks=callbacks_list,
                      validation_data=(exp_val_data,val_targets_c))
```

4.7.3 Results

Table 3 and Figure 14 show results achieved with networks of this slimmed-down architecture. With given *core kernel sizes* and *border kernel sizes*, all `mae` values stay completely unchanged (except some noise in the last digit).

But the same `mae` values now come with less costs, i.e. with less parameters to be trained: A `mae` magnitude of 10^{-4} that required 50 parameters in the initial simple convolutional network, and still 30 parameters in the previously improved network, now only needs just 25 parameters (*core kernel size=3, border kernel size=5*). That's an 50% improvement.

The ratio becomes even better if higher precision is required: A `mae` value of around $5 \cdot 10^{-7}$ required a whopping 100 parameters in the initial simple convolutional network, and still 50 parameters in the previously improved network. Now, only 39 parameters are needed to achieve the same result; an improvement of 61% .

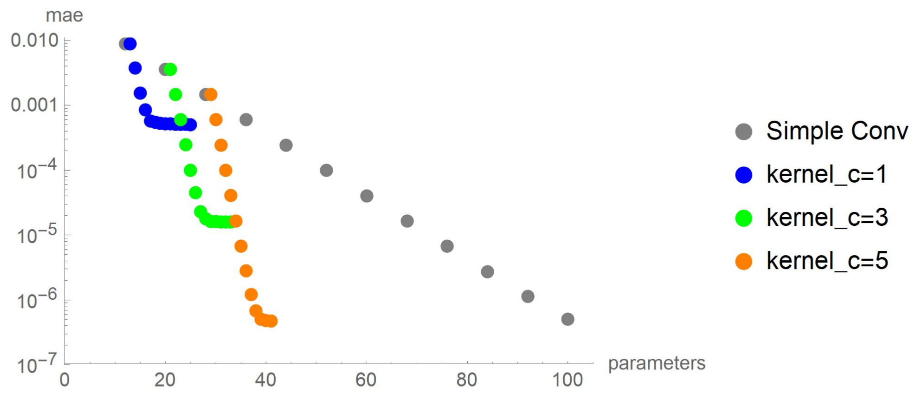

Figure 14: Networks with further slimmed-down, improved convolutional architecture compared to simple convolutional networks with different kernel sizes. For any slimmed-down network with a given *core kernel size* (`kernel_c`), the *border kernel size* has been varied between 1 and 13.

Core Kernel Size	**Border Kernel Size**	**Parameters**	**mae**
1	1	13	0.008837745
	2	14	0.003734339
	3	15	0.001531524
	4	16	0.000854901
	5	17	0.000572028
	6	18	0.000547835
	7	19	0.000523866
	8	20	0.000523113
	9	21	0.000517457
	10	22	0.000514886
	11	23	0.000511783
	12	24	0.000509274
	13	25	0.000506693
3	1	21	0.003594639
	2	22	0.001472260
	3	23	0.000599506
	4	24	0.000246431
	5	25	0.000100421
	6	26	0.000044958
	7	27	0.000023021
	8	28	0.000017994
	9	29	0.000016279
	10	30	0.000016204
	11	31	0.000016058
	12	32	0.000016041
	13	33	0.000015984
5	1	29	0.001465992
	2	30	0.000599010
	3	31	0.000244477
	4	32	0.000099736
	5	33	0.000040655
	6	43	0.000016612
	7	35	0.000006757
	8	36	0.000002813
	9	37	0.000001209
	10	38	0.000000680
	11	39	0.000000509
	12	40	0.000000484
	13	41	0.000000475

Table 3: Mean absolute error over a test set of 150,000 records produced by convolutional networks with further slimmed-down, improved architecture, using various core and border kernel sizes (each trained with 1,000,000 training records and 200,000 validation records). The task was to create 4 smoothly distributed sub-charges each, for a total of 32 original (coarse) charges.

5 Conclusions

In [Gelfand et al, 2016], a numerical simulation of the early stages of heavy-ion collisions in 3+1 dimensions is presented. In this simulation, there is a one-dimensional line of Wigner-Seitz cells, where each cell contains an individual total charge. For the simulation not to produce heavy numerical artifacts, the total charge in each Wigner-Seitz cell must be split into smaller sub-charges, which are then smoothly distributed as to approximate a continuous charge distribution. In this context, "smooth" means that the discrete fourth derivate should be constant within each cell. In [Gelfand et al, 2016], a (rather slow) iterative algorithm was implemented to simulate such a charge distribution.

In this paper, we have demonstrated that simple, dense neural networks with just one input layer, one output layer, and no bias weights, can be trained to learn this charge distribution task. These networks perform best with a linear activation function.

We have further derived a linear algorithm for calculating the exact charge distribution without neural networks, and presented a C++ implementation of this algorithm. This implementation works roughly three magnitudes faster than the algorithm used in [Gelfand et al, 2016]. Also, we have demonstrated that the weight matrices of the trained dense networks described above are equivalent to the exact solution.

We have demonstrated that the use of convolutional neural networks still should be considered when dealing with a very large number of charges. This is because in cases where the number of original charges becomes very large, the algorithm becomes more and more time consuming in its initialization phase. Also, for large numbers of charges, the algorithm for the exact solution performs a lot of unnecessary calculations because of the following fact:

Let Q_j be the single original charge in the j-th Wigner-Seitz cell, and let n be the number of (smaller) sub-charges to be created for each original charge Q_j. Then the original charge Q_j will be replaced by charges $q_\alpha \dots q_\omega$, with $\alpha = (Q-1)n+1$ and $\omega = nj$. Although in principle *all* charges $Q_1 \dots Q_{j_{max}}$ influence the values of $q_\alpha \dots q_\omega$, the actual influence of a specific charge Q_k on the values of $q_\alpha \dots q_\omega$ becomes smaller and smaller the farther away Q_k is from Q_j.

Therefore, for calculating $q_\alpha \dots q_\omega$ in good approximation, we can safely establish a method only considering a certain number of (left and right) neighboring charges of Q_j, and this is just what a convolutional network does. Hence, convolutional networks can calculate charge distributions for a virtually unlimited number of original charges in good approximation.

Finally, we have presented two more refined convolutional network architectures, by which the number of trainable parameters can be further reduced significantly.

6 Appendix

6.1 Listing "Charge Refine Train Data Generator"

```
"""
CHARGE REFINE TRAIN DATA GENERATOR
BY HELMUT HOERNER
V 1.0, (C) 2019
"""

import numpy as np
trainSetSize=2000000

numCells = 8  # number of Cells
pointsPerCell = 4 # number of sub-charges per cell
# absolute maximum deviation per sub-charge
maxErr=1E-05 / pointsPerCell

# **********************************
# Charge Generator
# **********************************
def chrg_generator(n):
    """ Creates test set with n entries """
    for i in range(n):
        # Creates random full charges
        Q=np.random.rand(numCells)
        # split full charges into pointsPerCell small charges per cell
        q=np.reshape([np.full(pointsPerCell,i) for i in Q],
                     (pointsPerCell*numCells))

        # First refinement
        while (not deviationOK(q,1)):
            j=0
            while j<50*pointsPerCell*numCells:
                j+=1
                refine(q,1)

        # Second refinement
        while (not deviationOK(q,2)):
            j=0
            while j<50*pointsPerCell*numCells:
                j+=1
                refine(q,2)
        yield([i-1, q, Q])
# **********************************
# Check Deviations
# **********************************
def deviationOK(q, step):
    """ check if there are deviations beyond maxErr """
    OK=True
    for i in range(0, numCells*pointsPerCell):
        if abs(dq_func(q, i, step))>maxErr:
            OK=False
            break
    return(OK)

# *****************************************
# single random charge refinement function
# *****************************************
def refine(chrg, step, i=-1):
    """ refines two neighboring charges chrg[i] and chrg[i+1]
    step ... refinement step 1 or 2 (-1 for random)
    i ... Index of charge to be refined (-1 for random)
    """

    if i==-1:
        i = np.random.randint(pointsPerCell*numCells)
    dq=dq_func(chrg, i, step)
    if dq!=0:
        chrg[i]-=dq
        chrg[i+1]+=dq
    return()
```

```
# **********************************
# dq_func
# **********************************
def dq_func(chrg, i, step):
    """ returns delta q, the amount by which charge chrg[i+1] must be changed
        and charge chrg[i] must be changed into the other direction
        Parameters:
        chrg ... Array of sub-charges
        i    ... index of charge to be modified (toegther with i+1)
        step ... 1 for first refinement, 2 for second refinement
        """
    dq=0.
    if (i+1)%pointsPerCell!=0:
        # modify only if not a charge on the right side of a cell
        if step==1:
            # First refinement:
            # Don't modify outermost charges
            if (i>0 and i<pointsPerCell*numCells-2):
                # modify charges
                dq=(chrg[i+2]-3*chrg[i+1]+3*chrg[i]-chrg[i-1])/4.
        elif step==2:
            # second refinement
            # Don't modify outermost charges
            if (i>1 and i<pointsPerCell*numCells-3):
                # modify charges
                dq=(-chrg[i+3]+5*chrg[i+2]-10*chrg[i+1]+
                    10*chrg[i]-5*chrg[i-1]+chrg[i-2])/12.
    return(dq)


# *************************************
# MAIN PROGRAM
# *************************************

# *************************************
# Generate train_data and train_targets
# *************************************
train_data=np.zeros((trainSetSize,numCells))
train_targets=np.zeros((trainSetSize,pointsPerCell*numCells))
k=0
for i, q, Q in chrg_generator(trainSetSize):
    if i%100==0:
        print("Computing TrainSet",i,"-",i+100)
    train_data[i]=Q
    train_targets[i]=q

# **********************
# Re-Scale train_targets
# **********************
ttmax=1.5
ttmin=-0.5
train_targets -= ttmin
train_targets /= (ttmax-ttmin)

# ****************
#  Pickle Data
# ****************
import os
import pickle
data_dir='F:\AI'
data_dir = os.path.join(data_dir,'ChargeRefine')
fname=os.path.join(data_dir,'train_data.pkl')
myFile=open(fname,"wb")
pickle.dump(train_data,myFile)
myFile.close()
fname=os.path.join(data_dir,'train_targets.pkl')
myFile=open(fname,"wb")
pickle.dump(train_targets,myFile)
myFile.close()
```

6.2 Listing "Charge Refine Deep Learning Explorer"

```
"""
CHARGE REFINE DEEP LEARNING EXPLORER
BY HELMUT HOERNER
V 1.0, (C) 2019
"""

import matplotlib.pyplot as plt
import os
import pickle
from keras import models
from keras import layers
from keras import callbacks

trainSetSize=1500000 # training data
valSetSize=300000 #validation data
testSetSize=200000 #test data
bSize=5000 # batch Size
myoptimizer='rmsprop' # optimizer
maxepochs=2000 # max number of epochs
mypat=2000 # 100 # stop after this no of epochs if no improvement

numCells = 8  # Number of Cells
pointsPerCell = 4 # Number of sub-charges per cell

# y positions of the full and split charges (not yet refined)
Qy=[i*pointsPerCell+float(pointsPerCell)/2.-0.5 for i in range(0, numCells)]
qy=[i for i in range(0, numCells*pointsPerCell)]
# x positions of cell borders
cellBorders=[i*pointsPerCell-0.5 for i in range(0, numCells+1)]

# **********************************
# Create Model
# **********************************
def createModel(nh1, nh2, nh3, nactfunc, actfuncin=True, actfuncout=False):
    if(nactfunc==0):
        actfunc='linear'
    elif(nactfunc==1):
        actfunc='sigmoid'
    elif(nactfunc==2):
        actfunc='tanh'
    elif(nactfunc==3):
        actfunc='softmax'
    elif(nactfunc==4):
        actfunc='elu'
    elif(nactfunc==5):
        actfunc='selu'
    elif(nactfunc==6):
        actfunc='relu'
    elif(nactfunc==7):
        actfunc='softplus'
    elif(nactfunc==8):
        actfunc='sigmoid'
    elif(nactfunc==9):
        actfunc='hard_sigmoid'
    elif(nactfunc==10):
        actfunc='exponential'

    model=models.Sequential()

    if actfuncin:
        model.add(layers.Dense(numCells, activation=actfunc,
                               input_dim=train_data.shape[1]))
    else:
        model.add(layers.Dense(numCells, input_dim=train_data.shape[1]))

    if nh1>0:
        model.add(layers.Dense(nh1, activation=actfunc))
    if nh2>0:
        model.add(layers.Dense(nh2, activation=actfunc))
    if nh3>0:
        model.add(layers.Dense(nh3, activation=actfunc))

    if actfuncout:
        model.add(layers.Dense(pointsPerCell*numCells, activation=actfunc))
    else:
        model.add(layers.Dense(pointsPerCell*numCells))
    return(model)
```

```
# **********************************
# Plot learning curve to file
# **********************************
def plotToFile(history, round, step):
    if(step==1):
        start=0
    elif(step==2):
        start=10
    elif(step==3):
        start=40
    elif(step==4):
        start=150

    loss=history.history['loss']
    val_loss=history.history['val_loss']
    epochs=range(1,len(loss)+1)
    plt.plot(epochs[start:], loss[start:], 'bo', label='Training loss')
    plt.plot(epochs[start:], val_loss[start:], 'b', label='Validation loss')
    plt.title('Training and Validation Loss')
    plt.legend()
    plt.ylabel("mean absolute error")
    plt.xlabel("epochs")
    imgfname="img"+str(step)+"_round"+str(round)+".png"
    imgfname=os.path.join(data_dir,imgfname)
    plt.savefig(imgfname, bbox_inches='tight', dpi=300)
    plt.clf()

# **********************************
# MAIN PROGRAM
# **********************************

# **********************************
# Load train_data and val_data
# **********************************

data_dir='F:\AI'
data_dir = os.path.join(data_dir,'ChargeRefine')

fname=os.path.join(data_dir,'train_data.pkl')
print("Loading ",fname)
myFile=open(fname,"rb")
file_train_data=pickle.load(myFile)
myFile.close()

fname=os.path.join(data_dir,'train_targets.pkl')
print("Loading ",fname)
myFile=open(fname,"rb")
file_train_targets=pickle.load(myFile)
myFile.close()

# extract train/val/test set
train_data=file_train_data[0:trainSetSize]
train_targets=file_train_targets[0:trainSetSize]

val_data=file_train_data[trainSetSize:trainSetSize+valSetSize]
val_targets=file_train_targets[trainSetSize:trainSetSize+valSetSize]

test_data=file_train_data[trainSetSize+valSetSize:
    trainSetSize+valSetSize+testSetSize]
test_targets=file_train_targets[trainSetSize+valSetSize:
    trainSetSize+valSetSize+testSetSize]

# **********************************
# Write Log File Header
# **********************************
fname=os.path.join(data_dir,'log.txt')
myFile=open(fname,"w")
myFile.write("trainSetSize="+str(trainSetSize)+", ")
myFile.write("valSetSize="+str(valSetSize)+", ")
myFile.write("testSetSize="+str(testSetSize)+", ")
myFile.write("numCells="+str(numCells)+", ")
myFile.write("pointsPerCell="+str(pointsPerCell)+", ")
myFile.write("Batch_size="+str(bSize)+", ")
myFile.write("Patience="+str(mypat)+", ")
myFile.write("Optimizer="+myoptimizer)
myFile.write("\n")

line="round, nh1, nh2, nh3, nactfunc, actfouncout, val_loss, test_loss, "
line+="epochs, BestSoFar\n"
myFile.write(line)
myFile.close()
```

```
# **********************************
# Define models to be tested
# Parameters:
# neurons hidden layer 1
# neurons hidden layer 2
# neurons hidden layer 3
# activation function 0...8
# activation function on input layer
# activation function on output layer
# **********************************
mylist=[]
mylist.append([16, 32, 16, 1, True, False])
mylist.append([16, 0, 0, 2, True, False])
mylist.append([0, 0, 0, 0, False, False])
# add more as needed

# *********************************
# Execute Simulations
# *********************************
print("Start Simulation")

callbacks_list = [
        callbacks.EarlyStopping(monitor='val_loss', patience=mypat,)
        ]
round=0
best_test_loss=999999

for nh1, nh2, nh3, nactfunc, actfuncin, actfuncout in mylist:
    round+=1
    print("**************************************************")
    print("ROUND ",round,nh1,nh2,nh3,nactfunc, actfuncin, actfuncout)
    print("**************************************************")
    # Simulation
    model=createModel(nh1,nh2,nh3,nactfunc,actfuncin, actfuncout)
    model.compile(optimizer=myoptimizer, loss='mae',
                  metrics=['mae'])
    history=model.fit(train_data,train_targets,
                      epochs=maxepochs, batch_size=bSize,
                      callbacks=callbacks_list,
                      validation_data=(val_data,val_targets))
    loss=history.history['loss']
    val_loss=history.history['val_loss']
    epochscount=len(loss)
    test_score=model.evaluate(test_data, test_targets)
    print ("Test Loss ", test_score[0])
    # write to log file
    line=str(round)+", "
    line+=str(nh1)+", "+str(nh2)+", "+str(nh3)+", "
    line+=str(nactfunc)+", "
    line+=str(actfuncin)+", "
    line+=str(actfuncout)+", "
    line+=str(val_loss[-1])+", "
    line+=str(test_score[0])+", "
    line+=str(epochscount)+", "

    if test_score[0]<best_test_loss:
        best_test_loss=test_score[0]
        print ("*********************************************")
        print ("Best so far "+line)
        print ("*********************************************")
        line+="*****"
    else:
        line+=" "
    line+="\n"
    myFile=open(fname,"a")
    myFile.write(line)
    myFile.close()

    # ************************************************
    # Plotting the results
    # ************************************************
    plotToFile(history,round,1)
    plotToFile(history,round,2)
    plotToFile(history,round,3)
    plotToFile(history,round,4)
```

6.3 Listing "Charge Refinement Class"

6.3.1 Header File

```
#define _SCL_SECURE_NO_WARNINGS
#pragma once
#include <iostream>
#include <vector>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;

/**************************************
CHARGE REFINEMENT CLASS HEADER
BY HELMUT HOERNER
VIENNA UNIVERSITY OF TECHNOLOGY
INTITUTE FOR THEORETICAL PHYSICS
V 0.1, (C) 2019
**************************************/

class clsChargeDistr
{
private:
  int chargeCount; // number of (original) charges
  int subChargesPerCell; // split-factor (number of "fine" charges per original charge)

  VectorXd charges; // vector with original (coarse) charge distribution
  double* charges_cpparr; // C++ array for handing over original (coarse) charge distribution
  vector<double> charges_vec; // vector object for handing over original charge distribution
  int charges_cpparrSize; // actual size (reserved memory) of cpp-array

  double* chargesRef1_cpparr; // C++ array for handing over first order solution
  double* chargesRef2_cpparr; // C++ array for handing over second order solution
  int chargesRef1_cpparrSize; // actual size (reserved memory) of cpp-array
  int chargesRef2_cpparrSize; // actual size (reserved memory) of cpp-array
  bool chargesRef1_arrvalid; // true, if chargesRef1_cpparr holds current values
  bool chargesRef2_arrvalid; // true, if chargesRef1_cpparr holds current values

  vector<double> chargesRef1_vec; // vector for handing over first order solution
  vector<double> chargesRef2_vec; // vector for handing over second order solution
  bool chargesRef1_vecvalid; // true, if chargesRef1_vec holds current values
  bool chargesRef2_vecvalid; // true, if chargesRef2_vec holds current values

  MatrixXd M1; // matrix for calulating first order solution
  MatrixXd M2; // matrix for calulating second order solution

private:
  void init();
  void invalidate();

public:
  clsChargeDistr();
  clsChargeDistr(int, int);
  ~clsChargeDistr();
  bool Prepare(int, int);
  double* getChargeArray();
  bool setChargeArray(double[]);
  bool setChargeVector(vector<double>&);
  vector<double>& getChargeVector();
  void setSingleCharge(int, double);
  double getSingleCharge(int);

  double getRefinedCharge(int, int);
  double* getRefinedChargeArray(int);
  vector<double>& getRefinedChargeVector(int);

  int getChrgCount();
  int getDistrChrgCount();
  int getSplitFactor();

  double getM1cell(int, int);
  double getM2cell(int, int);

};
```

6.3.2 clsChargeDistr

```
#include "clsChargeDistr.h"

/**************************************
CHARGE REFINEMENT CLASS
BY HELMUT HOERNER
VIENNA UNIVERSITY OF TECHNOLOGY
INTITUTE FOR THEORETICAL PHYSICS
V 0.1, (C) 2019
**************************************/

/**************************************
Empty Constructor
***************************************/
clsChargeDistr::clsChargeDistr()
{
  init();
  return;
}

/**************************************
Constructor with Prepare(..) call
***************************************/
clsChargeDistr::clsChargeDistr(int NoOfCharges, int SplitFactor)
{
  init();
  Prepare(NoOfCharges, SplitFactor);
  return;
}

/**************************************
Private helper method for constructors
***************************************/
void clsChargeDistr::init()
{
  chargeCount = 0;
  subChargesPerCell = 0;
  invalidate();
  charges_cpparrSize = 0;
  chargesRef1_cpparrSize = 0;
  chargesRef2_cpparrSize = 0;
}

/**************************************
Private helper method: invalidates results
***************************************/
void  clsChargeDistr::invalidate()
{
  chargesRef1_arrvalid = false;
  chargesRef2_arrvalid = false;
  chargesRef1_vecvalid = false;
  chargesRef2_vecvalid = false;
}

/**************************************
Destructor
***************************************/
clsChargeDistr::~clsChargeDistr()
{
  if (charges_cpparrSize > 0)
    delete[] charges_cpparr;
  if (chargesRef1_cpparrSize > 0)
    delete[] chargesRef1_cpparr;
  if (chargesRef2_cpparrSize > 0)
    delete[] chargesRef2_cpparr;
}
```

```cpp
/*************************************************
bool setChargeArray(double Arr[])
sets original (coarse) charge distribution
Prepare must be called first
Arr[] .. Array with the charges before refinement
Returns false on error
*************************************************/
bool clsChargeDistr::setChargeArray(double Arr[])
{
  if (chargeCount == 0)
  {
    cerr << "Error! Must call Prepare before calling setChargeArray!" << endl;
    return(false);
  }
  // copy Arr to local array
  copy(Arr, Arr + chargeCount, charges_cpparr);
  // map local array to Eigen-vector
  charges = Map<VectorXd>(charges_cpparr, chargeCount);
  // all former calculations are invalid
  invalidate();
  return(true);
}

/*************************************************
bool setChargeVector(vector<double> &vec)
sets original (coarse) charge distribution
Prepare must be called first
&vec .. Vector with charges before refinement
Returns false on error
*************************************************/
bool clsChargeDistr::setChargeVector(vector<double> &vec)
{
  if (chargeCount == 0)
  {
    cerr << "Error! Must call Prepare before calling setChargeArray!" << endl;
    return(false);
  }
  // copy vector to local array
  copy(vec.begin(), vec.end(), charges_cpparr);
  // map local array to Eigen vector
  charges = Map<VectorXd>(charges_cpparr, chargeCount);
  // all former calculations are invalid
  invalidate();
  return(true);
}

/*************************************************
double* getChargeArray()
Returns original (coarse) charge distribution
Prepare must be called first
*************************************************/
double* clsChargeDistr::getChargeArray()
{
  if (chargeCount == 0)
  {
    cerr << "Error! Must call Prepare before calling getChargeArray!" << endl;
    return(0);
  }
  // map charge vector to local array
   Map<MatrixXd>(charges_cpparr, charges.rows(), charges.cols()) = charges;
  // return array
  return(charges_cpparr);
}
```

```
/*************************************************
double* getChargeVector()
Returns original (coarse) charge distribution
Prepare must be called first
*************************************************/
vector<double>& clsChargeDistr::getChargeVector()
{

  charges_vec.clear();
  if (chargeCount == 0)
  {
    cerr << "Error! Must call Prepare before calling getChargeVector!" << endl;
    return(charges_vec);
  }
  // copy charge vector to local array
  Map<MatrixXd>(charges_cpparr, charges.rows(), charges.cols()) = charges;
  // copy local array to local charge vector
  charges_vec.insert(charges_vec.end(), &charges_cpparr[0], &charges_cpparr[chargeCount]);
  // return vector by reference
  return(charges_vec);
}

/*******************************************
setSingleCharge(int index, double val)
Sets the value of a single (coarse) original charge
Prepare(NoOfCharges,SplitFactor) must be called first
index ... between 0 and NoOfCharges-1
val ... charge value
*******************************************/
void clsChargeDistr::setSingleCharge(int index, double val)
{
  if (index < 0 || index >= chargeCount)
    cerr << endl << "Error! Invalid index in method setSingleCharge!" << endl;
  else
  {
    charges(index) = val;
    invalidate();
  }
}

/*******************************************
getSingleCharge(int index)
Returns single (original) charge value
*******************************************/
double clsChargeDistr::getSingleCharge(int index)
{
  if (index < 0 || index >= chargeCount)
    return(0.0);
  else
    return(charges(index));
}
```

```
/**************************************************
Prepare(int NoOfCharges, int SplitFactor)
Creates charge distribution matrices M1 and M2
NoOfCharges ... number of (coarse) original charges
SplitFactor ... number of refined charges per
original charge
Returns false on error
**************************************************/
bool clsChargeDistr::Prepare(int NoOfCharges, int SplitFactor)
{
  int i;
  int j;
  // all former calulations are invalid
  invalidate();

  if (NoOfCharges < 3)
  {
    cerr << "Error! Number of charges in method Prepare must be larger than 2." << endl;
    return(false);
  }

  if (SplitFactor <= 1)
  {
    cerr << "Error! Split-factor in method Prepare must be larger than 1." << endl;
    subChargesPerCell = 0;
    return(false);
  }

  // Allocate additional memory for C++ handover arrays, if neccessary
  if (NoOfCharges > charges_cpparrSize)
  {
    if(charges_cpparrSize>0)
      delete[] charges_cpparr;
    charges_cpparr = new double[NoOfCharges];
    charges_cpparrSize = NoOfCharges;
  }
  if (NoOfCharges*SplitFactor > chargesRef1_cpparrSize)
  {
    if (chargesRef1_cpparrSize>0)
      delete[] chargesRef1_cpparr;
    chargesRef1_cpparr = new double[NoOfCharges*SplitFactor];
    chargesRef1_cpparrSize = NoOfCharges*SplitFactor;
  }
  if (NoOfCharges*SplitFactor > chargesRef2_cpparrSize)
  {
    if (chargesRef2_cpparrSize>0)
      delete[] chargesRef2_cpparr;
    chargesRef2_cpparr = new double[NoOfCharges*SplitFactor];
    chargesRef2_cpparrSize = NoOfCharges*SplitFactor;
  }
  for (i = 0;i < NoOfCharges;i++)
    charges_cpparr[i] = double(0.);

  chargeCount = NoOfCharges;
  subChargesPerCell = SplitFactor;

  // ------------------------------
  // Calculate first order Matrix M1
  // ------------------------------
  MatrixXd A;
  int dcCount = getDistrChrgCount();

  // Generate inital matrix representing EQ system
  A = MatrixXd::Zero(dcCount, dcCount);
  for (i = 0; i < dcCount; i++)
  {
    if (i % subChargesPerCell == 0)
    {
      // every subChargesPerCell line: equation
      // "sum of distrcharges == total charge"
      for(j=i;j<i+subChargesPerCell;j++)
        A(i, j) = double(1);
    }
    else if(i==1)
    {
      // special treatment for second row
      // q1 = Q1/subChargesPerCell
      A(i, 0) = double(subChargesPerCell);
    }
    else if (i == dcCount-1)
    {
      // special treatment for last row
      // q_last = Q_last/subChargesPerCell
      A(i, i) = double(subChargesPerCell);
    }
    else
    {
      // general first order distr. equation
      A(i, i - 2) = double(0.5);
      A(i, i - 1) = double(-1.5);
      A(i, i) = double(1.5);
      A(i, i + 1) = double(-0.5);
    }
  }

  // Invert matrix A
  MatrixXd I = MatrixXd::Identity(dcCount, dcCount);
  MatrixXd Ainv(dcCount, dcCount);
  Ainv = A.householderQr().solve(I);

  // Fill in sure zeros to minimize error
  // first row zeros
  Ainv(0, 0) = double(0.);
  for (i = 2; i < dcCount; i++)
    Ainv(0, i) = double(0.);
  // last row zeros
  for (i = 0; i < dcCount-1; i++)
    Ainv(dcCount-1, i) = double(0.);

  // Calculate final matrix
  M1 = MatrixXd::Zero(dcCount, chargeCount);
  VectorXd Q;
  for(i = 0; i < chargeCount; i++)
  {
    // initialize charge vector Q for i-th charge
    Q = VectorXd::Zero(dcCount);
    Q(i*subChargesPerCell) = 1;
    if (i == 0)
      Q(1) = 1;
    if (i == chargeCount -1)
      Q(dcCount-1) = 1;

    // Calculate i-th column of solution matrix
    M1.col(i) = Ainv*Q;
  }

  // ---------------------------------
  // Calculate second order Matrix M2
  // ---------------------------------
  // Generate inital matrix representing EQ system
  A = MatrixXd::Zero(dcCount, dcCount);
  for (i = 0; i < dcCount; i++)
  {
    if (i % subChargesPerCell == 0)
    {
      // every subChargesPerCell line: equation
      // "sum of distrcharges == total charge"
      for (j = i;j<i + subChargesPerCell;j++)
        A(i, j) = double(1);
    }
    else if (i == 1)
    {
      // special treatment for second row
      // q_0 = Q_0/subChargesPerCell
      A(i, 0) = double(subChargesPerCell);
    }
    else if (i == 2)
    {
      // special treatment for third row
      // q_1 = result from M1
      A(i, 1) = double(1.);
    }
    else if (i == dcCount - 2)
    {
      // special treatment for row before last
      // q_secondlast = result from M1
      A(i, i) = double(1.);
    }
    else if (i == dcCount - 1)
    {
      // special treatment for last row
      // q_last = Q_last/subChargesPerCell
      A(i, i) = double(subChargesPerCell);
    }
    else
    {
      // general second order distr. equation
      A(i, i - 3) = double(0.5);
      A(i, i - 2) = double(-2.5);
      A(i, i - 1) = double(5);
      A(i, i) = double(-5);
      A(i, i + 1) = double(2.5);
      A(i, i + 2) = double(-0.5);
    }
  }

  // Invert matrix A
  Ainv = A.householderQr().solve(I);

  // Fill in sure zeros to minimize error
  // first row zeros
  Ainv(0, 0) = double(0.);
  for (i = 2; i < dcCount; i++)
    Ainv(0, i) = double(0.);
  //second row zeros
  Ainv(1, 0) = double(0.);
  Ainv(1, 1) = double(0.);
  for (i = 3; i < dcCount; i++)
    Ainv(1, i) = double(0.);
  // second last row zeros
  for (i = 0; i < dcCount - 2; i++)
    Ainv(dcCount - 2, i) = double(0.);
  Ainv(dcCount - 2, dcCount - 1) = double(0.);
  // last row zeros
  for (i = 0; i < dcCount - 1; i++)
    Ainv(dcCount - 1, i) = double(0.);

  // Calculate final matrix
  M2 = MatrixXd::Zero(dcCount, chargeCount);

  for (i = 0; i < chargeCount; i++) //
  {
    // initialize charge vector Q for i-th charge
    Q = VectorXd::Zero(dcCount);
    Q(2) = M1(1,i);
    Q(dcCount -2) = M1(dcCount - 2, i);
    Q(i*subChargesPerCell) = 1;
    if (i == 0)
      Q(1) = 1;
    if (i == chargeCount - 1)
      Q(dcCount - 1) = 1;

    // Calculate i-th column of solution matrix M2
    M2.col(i) = Ainv*Q;
  }

  charges = Map<VectorXd>(charges_cpparr, chargeCount);
  return(true);

}

/************************************************
getChrgCount()
Returns the number of (coarse) original charges
************************************************/
int clsChargeDistr::getChrgCount()
{
  return(chargeCount);
}

/************************************************
getDistrChrgCount()
Returns the number of (fine) distributed charges
************************************************/
int clsChargeDistr::getDistrChrgCount()
{
  return(chargeCount*subChargesPerCell);
}

/************************************************
getSplitFactor()
Returns the split factor (No of subChargesPerCell)
************************************************/
int clsChargeDistr::getSplitFactor()
{
  return(subChargesPerCell);
}
```

```
/*************************************************
getM1cell(int row, int col)
Returns cell of M1 matrix
*************************************************/
double clsChargeDistr::getM1cell(int row, int col)
{
  if (row < 0 || col < 0)
    return(double(0.));

  if (row >= getDistrChrgCount() || col >= chargeCount)
    return(double(0.));

  return(M1(row,col));
}

/*************************************************
getM2cell(int row, int col)
Returns cell of M2 matrix
*************************************************/
double clsChargeDistr::getM2cell(int row, int col)
{
  if (row < 0 || col < 0)
    return(double(0.));

  if (row >= getDistrChrgCount() || col >= chargeCount)
    return(double(0.));

  return(M2(row, col));
}

/*************************************************
double* getRefinedChargeArray(int order)
Returns refined charge distribution as C++ array
order ... 1 or 2 (everything else intepreted as 2)
*************************************************/
double* clsChargeDistr::getRefinedChargeArray(int order)
{
  VectorXd Solution;
  if (order == 1)
  {
    if (!chargesRef1_arrvalid)
    {
      Solution= M1*charges;
      Map<MatrixXd>(chargesRef1_cpparr, Solution.rows(), Solution.cols()) = Solution;
      chargesRef1_arrvalid = true;
    }
    return(chargesRef1_cpparr);
  }
  else
  {
    if (!chargesRef2_arrvalid)
    {
      Solution = M2*charges;
      Map<MatrixXd>(chargesRef2_cpparr, Solution.rows(), Solution.cols()) = Solution;
      chargesRef2_arrvalid = true;
    }
    return(chargesRef2_cpparr);
  }
}
```

```
/**************************************************
vector<double>& getRefinedChargeVector(int order)
Returns refined charge distribution as vector
order ... 1 or 2 (everything else intepreted as 2)
**************************************************/
vector<double>& clsChargeDistr::getRefinedChargeVector(int order)
{
  if (order == 1 && chargesRef1_vecvalid)
    return(chargesRef1_vec);

  if (order != 1 && chargesRef2_vecvalid)
    return(chargesRef2_vec);

  getRefinedChargeArray(order);
  if (order == 1)
  {
    chargesRef1_vec.clear();
    chargesRef1_vec.insert(chargesRef1_vec.end(), &chargesRef1_cpparr[0], &chargesRef1_cpparr[
     chargeCount*subChargesPerCell]);
    chargesRef1_vecvalid = true;
    return(chargesRef1_vec);
  }
  else
  {
    chargesRef2_vec.clear();
    chargesRef2_vec.insert(chargesRef2_vec.end(), &chargesRef2_cpparr[0], &chargesRef2_cpparr[
     chargeCount*subChargesPerCell]);
    chargesRef2_vecvalid = true;
    return(chargesRef2_vec);
  }
}

/**************************************************
double getRefinedCharge(int order, int index)
Returns the value of a single refined charge
order ... 1 or 2 (everything else intepreted as 2)
index .. between 0 and getDistrChrgCount()
**************************************************/
double clsChargeDistr::getRefinedCharge(int order, int index)
{
  if (index < 0 || index >= chargeCount*subChargesPerCell)
    return(double(0));

  return(getRefinedChargeArray(order)[index]);
}
```

6.4 Convolutional Network with Handling of Boundary Charges

```
"""
CHARGE REFINEMENT BY CONVOLUTIONAL NETWORK
WITH ADDITIONAL BOUNDARY HANDLING
V 1.0, (C) 2019 HELMUT HOERNER
"""

import os
import numpy as np
import matplotlib.pyplot as plt
from keras import models
from keras import layers
from keras import callbacks

QCount=32;
SplitFactor=4;
FileName = 'Matrix2_'+str(QCount)+'_'+str(SplitFactor)+'.txt'

kernel=13
padding=int((kernel-1)/2) # cells left and right to be left out
np.random.seed(0) # make pseudo random numbers reproducible

maxepochs=10000 # max number of epochs
mypat=20 # stop after this no of epochs if no improvement
trainSetSize=1000000 # training data
valSetSize  = 200000 # validation data
testSetSize = 150000 #test data
bSize=250000 # batch Size
myoptimizer='Adam' # optimizer

# **********************************
# Charge Plotting Function
# **********************************
def pltChrge(Q, q, q_pred, lines=False, title='',legend=False):
    plt.figure(figsize=(16, 8), dpi=150)
    cellBorders=[i*SplitFactor-0.5 for i in range(0, Qcount+1)]
    # x positions of the full and split charges (not yet refined)
    Qx=[i*SplitFactor+float(SplitFactor)/2.-0.5 for i in range(0, Qcount)]
    qx=[i for i in range(0, Qcount*SplitFactor)]
    for xc in cellBorders:
        plt.axvline(x=xc, color='gray')

    plt.plot(Qx,Q,'ro', markersize=12, label='full charges')
    plt.plot(qx,q*SplitFactor,'bo',label='split charges*'+str(SplitFactor))
    plt.plot(qx,q_pred*SplitFactor,'go',label='predition*'+str(SplitFactor))

    if lines:
        plt.plot(qx,q*SplitFactor,'b')
        plt.plot(qx,q_pred*SplitFactor,'g')
    if legend:
        plt.legend()
    if title!='':
        plt.title(title)
    return()

# ********************************************
# Generate and load matrix M2
# ********************************************
if os.path.isfile(FileName):
    print(FileName,"already exists.")
else:
    print('Generating matrix file',FileName)
    os.system('GenM2.exe '+str(QCount)+' '+str(SplitFactor))

print('Load matrix file')
f=open(FileName,encoding="utf-8")
MatrixData=f.read()
f.close()
MatrixLines=MatrixData.split('\n')
Qcount = len(MatrixLines[0].split(','))
qcount = len(MatrixLines)
SplitFactor = int(qcount/Qcount)

# parsing matrix data
M2 = np.zeros((qcount, Qcount))
for i, line in enumerate(MatrixLines):
    sline=line.split(',')
    values = [float(x) for x in sline]
    M2[i,:]=values
```

```
# **********************************
# Charge Generator
# **********************************
def chrg_generator(n):
    """ Creates test set with n entries """
    for i in range(n):
        # Creates random full charges
        Q=np.random.rand(Qcount)
        q=M2@Q
        yield([i-1, q, Q])

# ************************************
# Generate data
# ************************************
totSize=trainSetSize+valSetSize+testSetSize
data=np.zeros((totSize,Qcount))
targets=np.zeros((totSize,qcount))
targets_c=np.zeros((totSize,qcount-2*padding*SplitFactor))
targets_l=np.zeros((totSize,padding*SplitFactor))
offset=0
for i, q, Q in chrg_generator(trainSetSize+valSetSize+testSetSize):
    if i%100000==0:
        print("Computing TrainSet",i,"-",i+99999)
    data[i]=Q
    targets[i]=q
    targets_c[i]=q[padding*SplitFactor:-padding*SplitFactor]
    targets_l[i]=q[:padding*SplitFactor]

#re-shape input data for conv network
exp_data=np.expand_dims(data, axis=2)

# Seperate Train Data
train_data=data[:trainSetSize]
exp_train_data=exp_data[:trainSetSize]
train_targets=targets[:trainSetSize]
train_targets_c=targets_c[:trainSetSize]
train_targets_l=targets_l[:trainSetSize]


# Seperate Validation Data
val_data=data[trainSetSize:trainSetSize+valSetSize]
exp_val_data=exp_data[trainSetSize:trainSetSize+valSetSize]
val_targets=targets[trainSetSize:trainSetSize+valSetSize]
val_targets_c=targets_c[trainSetSize:trainSetSize+valSetSize]
val_targets_l=targets_l[trainSetSize:trainSetSize+valSetSize]

# Seperate Test Data
test_data=data[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
exp_test_data=exp_data[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
test_targets=targets[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
test_targets_c=targets_c[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]
test_targets_l=targets_l[trainSetSize+valSetSize:
        trainSetSize+valSetSize+testSetSize]

callbacks_list = [
        callbacks.EarlyStopping(monitor='val_loss', patience=mypat,)]

# **********************************
# Create and train Model for left border
# **********************************
print("**************")
print("Train model for LEFT",padding," charges")
Lmodel=models.Sequential()
Lmodel.add(layers.Dense(kernel, activation='linear', use_bias=False,
                               input_dim=train_data[:,:kernel].shape[1]))
Lmodel.add(layers.Dense(padding*SplitFactor), use_bias=False)

Lmodel.compile(optimizer=myoptimizer, loss='mse', metrics=['mae'])
Lhistory=Lmodel.fit(train_data[:,:kernel],train_targets_l,
                     epochs=maxepochs, batch_size=bSize,
                     callbacks=callbacks_list,
                     validation_data=(val_data[:,:kernel],val_targets_l))
```

```
# ***********************************
# Create and train Model for core data
# ***********************************
print("**************")
print("Train core data model, exluding border 2 *",padding)
model=models.Sequential()
model.add(layers.Conv1D(filters=SplitFactor, use_bias=False,
                        kernel_size=kernel,
                        input_shape=(Qcount,1)))
model.add(layers.Flatten())

model.compile(optimizer=myoptimizer, loss='mse', metrics=['mae'])
history=model.fit(exp_train_data,train_targets_c,
                      epochs=maxepochs, batch_size=bSize,
                      callbacks=callbacks_list,
                      validation_data=(exp_val_data,val_targets_c))

test_score=model.evaluate(exp_test_data, test_targets_c)
print ("Test Loss ", test_score[0])

# ****************************************************
#  Plot prediction on first test set record
# ****************************************************
prediction=np.zeros(qcount) # empty array

# predict left charges with Lmodel
prediction[:padding*SplitFactor]=Lmodel.predict(test_data[0:1,:kernel])

# predict right charges also using Lmodel by reversing charge array
prediction[-padding*SplitFactor:]= \
Lmodel.predict(test_data[0:1,:Qcount-1-kernel:-1])[0,::-1]

# predict all other charges using core Conv1D model
prediction[padding*SplitFactor:-padding*SplitFactor]= \
  model.predict(exp_test_data[0:1])[0]

pltChrge(test_data[0], test_targets[0],
         prediction, True, '',True)
```

List of Tables

List of Figures

References

[Gelfand et al, 2016] D. Gelfand, A. Ipp, D. Müller, "Simulating collisions of thick nuclei in the color glass condensate framework", *Phys.Rev.* **D94** (2016) no.1, 014020 arXiv:1605.07184 [hep-ph] TUW-16-14

[Goodfellow et al, 2016] I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning", *MIT Press* (2016), Cambridge, Mass.

[Moore et al, 1998] G. D. Moore, C.-r. Hu, and B. Müller, "Chern-Simons number diffusion with hard thermal loops" *Phys.Rev.* **D58** (1998) 045001, arXiv:hep-ph/9710436