# Towards a general stabilisation method for conservation laws using a multilayer Perceptron neural network: 1D scalar and system of equations

Veiga, Maria Han ; Abgrall, Rémi

6th European Conference on Computational Mechanics (ECCM 6)
7th European Conference on Computational Fluid Dynamics (ECFD 7)
11-15 June 2018, Glasgow, UK

# Towards a general stabilisation method for conservation laws using a multilayer Perceptron neural network: 1D scalar and system of equations

**Maria Han Veiga**[1] **and Rémi Abgrall**[2]

[1] Institute for Computational Science and Institute für Mathematik, University of Zürich, Wintherthurerstrasse 190, hmaria@physik.uzh.ch
[2]Institute für Mathematik, University of Zürich, Wintherthurerstrasse 190, remi.abgrall@uzh.ch

**Keywords**: *limiters, computing methods, neural networks*

In this work we explore the idea of a parameter free stabilisation method for hyperbolic conservation laws. It is well known that for discontinuous solutions, non-physical oscillations will develop around the discontinuity. There are different methods to control these oscillations, namely, adding a viscosity term to the partial differential equation or using limiters to modify the solution.

In this work we show how to use a neural network to identify cells which are in need of stabilisation, which is parameter free and constructed to be independent of the underlying discretization scheme. We show the performance of this trouble cell indicator for a DG scheme for scalar (linear advection) and systems of equations (Euler equations) in one dimension.

## 1 Introduction

When dealing with nonlinear conservation laws (1), it is well known that discontinuous solutions can emerge, even for smooth initial data. The numerical approximation of discontinuous solutions will develop non-physical oscillations around the discontinuity, which in turn will negatively impact the accuracy of the numerical scheme. There are different methods to control these oscillations, namely, adding a viscous term as in (2) or limiting the solution.

$$\frac{\partial}{\partial t}u + \nabla \cdot f(u) = 0 \tag{1}$$

$$\frac{\partial}{\partial t}u + \nabla \cdot f(u) = \nabla \cdot (\nu(u)\nabla u) \tag{2}$$

Neural networks gained new popularity due to the computational tractability of the back-propagation algorithm, used for the learning of weights and biases in a deep neural network. Furthermore, it has been empirically shown to generate robust models for classification in many areas of applications [1, 2] and theoretically, to generate *universal* classifiers and *universal* function approximators [3, 4, 5].

We can write a stabilisation method as a function $f$ that takes some local properties of the solution (denoted by the map $X(u(x))$), and returns a modified solution value $\tilde{u}(x)$, which has some desired properties, e.g. non-oscillatory, maximum principle preserving.

$$f\big(u(x), X(u(x))\big) = \tilde{u}(x)$$

For example, in the context of finite volume schemes, we can write the **Minmod limiter** as:

$$f((u_j), X(u_j)) = u_j - \text{minmod}(X(u_j)) = \tilde{u}_j$$

where the map $X(u_j)$ encodes local properties of the solution at $u(x_j)$. In this case, $X(u_j) = (\bar{u}_j - u^+_{j-1/2}, \Delta^+\bar{u}_j, \Delta^-\bar{u}_j)$, where the usual notation applies: $\bar{u}$ the cell average, $\Delta^+\bar{u}_j := u_{j+1} - u_j$, $\Delta^-\bar{u}_j := u_j - u_{j-1}$ and $u_* = u(x_*)$, .

It is well known this clips the extrema for smooth solutions. To minimize this effect, one can consider a **TVD limiter** [6]:

$$f(u(x_j), X(u(x_j)), M) = u(x_j) - \text{TVD}(X(u(x_j)), M) = \tilde{u}_j$$

where the map $X(u(x_j)) = (\bar{u}_j - u^+_{j-1/2}, \Delta^+\bar{u}_j, \Delta^-\bar{u}_j)$ encodes local properties of the solution at $u(x_j)$, and

$$TVD(X(u_j), M) = \begin{cases} minmod(X(u_j)) & \text{else} \\ u(x_j) & \text{if } |\bar{u}_j - u^+_{j-1/2}| \leq M(\Delta x)^2, \end{cases}$$

where M is a user defined parameter and gives an estimate of the smoothness of the solution $u(x)$. It is to note that $M$ can take a value in a large range of positive numbers, and that it is usually a global quantity fixed in the beginning of the numerical experiment. Thus, this can be a drawback if the solution has different smoothness properties across the domain.

In the context of discontinuous Galerkin method, we can mention, among many, the high-order limiter by [7], which does the limiting in a hierarchical manner, relying on the modal representation of the solution $u(x)$. This does not clip the solution maxima, thus having desirable properties, but it is formulated specifically for the modal discontinuous Galerkin method.

In this work we learn a data-driven stabilisation function $f(X(u(x)))$ which doesn't require user input, and explicitly show to integrate such stabilisation method with a working 1-dimensional computational fluid dynamics (CFD) code. Furthermore, we design this stabilisation function with the intent to be independent of the underlying numerical scheme, so the input required is formulated by nodal values of the solution and finite differences.

The paper is structured as follows: in section 2 the construction of the dataset is described, the methodology in presented in section 3, numerical results are shown in section 4 and conclusions in chapter 5.

In the spirit of open science, all the codes and trained models are made available in a public repository [8].

## 2 Dataset

The dataset is an integral part of data-driven analysis. It contains the data for which we want to learn a mapping for. For our task, to learn a function which indicates whether a discontinuity is present in the solution or not, our data is the set containing $N$ samples $\{X_i, y_i\}_{i=1}^N$, where $X_i$ denotes some local properties of the solution $u_i$ (*features*) and $y_i$ (*labels*) indicates the existence (or not) of a discontinuity.

The dataset is generated by performing many runs on a 1-dimensional discontinuous Galerkin code solving the advection equation for different initial conditions, orders and mesh sizes (see table 1), and the labels are obtained by running a *high-order limiter* for discontinuous Galerkin [7] which does hierarchical limiting of the modes.

The interesting point of generating the labels using this is that we can learn the black box function of a limiter which is specific to a particular framework (in this case, discontinuous Galerkin) and integrate this with other numerical methods. Furthermore, the cell which are flagged by the *high-order limiter* are similar to a well tuned TVD limiter for approximation orders higher than 2.

| | |
|---|---|
| Initial condition | $\sin(2\pi x)$, $\begin{cases} 4 & 0.25 \leq x \leq 0.75 \\ 1 & \text{otherwise} \end{cases}$, $\begin{cases} \frac{1}{2}\sin(2\pi x) & x \leq 0.3 \\ 0 & \text{otherwise} \end{cases}$ $x \in [0, 1]$ |
| Mesh size | 8, 16, 32, 64, 128 |
| Order | 2, 3 |

**Table 1**: Runs used to generate the dataset.

### 2.1 Features

The features $X_i$ are the different quantities used to describe the local solution $u_i$. For the sake of generalisation, we choose features which are readily available in different numerical methods, such as: cell mean value, values at interface, divided differences between neighbours, etc. The complete description of the features can be found in table 2.

Of course, one could consider features which are specific for a particular method, e.g. the modes of the solution when using a discontinuous Galerkin method, or the smoothness parameters for a WENO method.

## 3 Method

In the following section, we focus on the three main aspects of this method:

1. the set-up of the learning algorithm;
2. the integration of a trained model with an existing CFD code;
3. the performance measures used to validate this method.

For the specific implementation details, please refer to the github repository [8].

| ID | Feature Name | Description |
|---|---|---|
| 1 | h | cell width |
| 2 | $\bar{u}_i$ | average value of solution at cell $i$ |
| 3 | $\bar{u}_{i+1}$ | average value of solution at cell $i+1$ |
| 4 | $\bar{u}_{i-1}$ | average value of solution at cell $i-1$ |
| 5 | $u^+_{i-\frac{1}{2}}$ | value of solution at interface $i-1/2$ reconstructed in cell $i$ |
| 6 | $u^-_{i+\frac{1}{2}}$ | value of solution at interface $i+1/2$ reconstructed in cell $i$ |
| 7 | $u^-_{i-\frac{1}{2}}$ | value of solution at interface $i-1/2$ reconstructed at cell $i-1$ |
| 8 | $u^+_{i+\frac{1}{2}}$ | value of solution at interface $i+1$ reconstructed at cell $i+1$ |
| 9 | $du_{i+1}$ | divided difference between $\bar{u}_i$ and $\bar{u}_{i+1}$, divided by $h$ |
| 10 | $du_{i-1}$ | divided difference between $\bar{u}_i$ and $\bar{u}_{i-1}$, divided by $h$ |
| 11 | $du_i$ | divided difference between $\bar{u}_{i+1}$ and $\bar{u}_{i-1}$, divided by $2h$ |

**Table 2**: Features table

### 3.1 Training a neural network offline

In this section we describe the details of the learning algorithm. We use a multilayer Perceptron neural network. For a general comprehensive introduction of neural networks one can refer to [9].

We wish to learn a map $f : \mathbf{X} \to \mathcal{Y}$, where $\mathbf{X}$ denotes an arbitrary set containing examples that we wish to label with possible outcomes $\mathcal{Y}$. The task at hand is a binary classification (i.e. does this cell require stabilisation?), thus $f$ will be our binary classifier and $\mathcal{Y} = \{0, 1\}$.

We choose $f$ to be defined by the composition of a sequence of functions $g_1, g_2, ...g_n$, yielding

$$f(x) = g_n(...g_2(g_1(x))).$$

This is known as a deep neural network. There are many different classifiers which can be used, but it has been shown that deep neural networks perform well on a variety of classification tasks, in particular when the classification plane is nonlinear [9].

Each function $g_i(w_i, b_i, h_i(\cdot))$ is parametrized by a matrix $w_i$, called weights, a vector $b_i$ called bias and an activation function $h_i(\cdot)$ which introduces the non-linearity on the neural network.

These parameters are tuned through a *loss function* $\mathcal{L}(x)$, which measures how well the mapping $f$ performs on a given dataset $\mathcal{D}$, using the gradient descent algorithm and back-propagation.

The gradient descent [10] is a first-order iterative optimization algorithm for finding the minimum of a function (in this case, the loss function $\mathcal{L}(x)$), relying on the fact that for small enough $\eta$:

$$a_{n+1} = a_n - \eta \nabla \mathcal{L}(a_n)$$

then $\mathcal{L}(a_{n+1}) \leq \mathcal{L}(a_n)$, for a defined and differentiable loss function $\mathcal{L}$. The size of the update on the opposite direction of the steepest gradient is tuned through $\eta$, the *learning rate*. Instead of using a global learning rate, we use the Adam algorithm [11] to adaptively estimate the update for each parameter (in this case, for the weights and biases), which uses information from the first and second moments of the gradient (with respect to

parameter $a$) $\nabla\mathcal{L}(a_n)$. The update follows:

$$a_{n+1} = a_n - \alpha\frac{\hat{m}_n}{\sqrt{\hat{v}_n} + \varepsilon},$$

where $\alpha$ denotes the *stepsize*, typically set to $\alpha = 0.001$, $\hat{m}_t$ denotes the *corrected* moving first moment of the gradient, $\hat{v}_t$ the *corrected* moving second moment of the gradient (both with respect to parameter $a$) and $\varepsilon$ a small value to avoid division by zero. For the full algorithm, please refer to appendix B.

Finally, the back-propagation algorithm is used, which gives a rule to update the different $w_i$ and $b_i$ in order to minimize the loss function [12].

Finally, two different loss functions are used:

- cross-entropy:

$$\mathcal{L}(\mathcal{D}) = -\frac{1}{N}\sum_i^N y_i \log(\hat{p}_i) + (1 - y_i)\log(1 - \hat{p}_i). \tag{3}$$

- weighted cross-entropy:

$$\mathcal{L}(\mathcal{D}) = -\frac{1}{N}\sum_i^N y_i \log(\hat{p}_i)\omega + (1 - y_i)\log(1 - \hat{p}_i). \tag{4}$$

One particular property of our problem is that it is expected that there will be a class imbalance on the dataset (both during the training phase and prediction phase). In particular, it is more likely to find cells which are in no need for stabilisation than ones which are in need for stabilisation. Furthermore, the cost of overlimiting is less than the cost of missing a cell that needs limiting, as it might lead to unphysical results and potentially crash the code. To bias the learning towards predicting the less represented class (needing stabilisation) better, it is common practice to use a weighted cost function[13]. The asymmetry in the loss function is added through the coefficient $\omega$.

Finally, we specify the activation functions used: for the hidden layers $1, ..., n-1$ a Rectified Linear Unit (ReLU) is used:

$$h(x) = \max(0, x).$$

Although there exist more sophisticated activation functions, typically modifications to ReLU, e.g. leaky ReLU, Parametric ReLU or Randomised leaky ReLU, these require further parameter estimation, add at least one more dependence to the saved model, and the empirical improvement on the performance is not extremely significant [14]. Furthermore, there are theoretical predictions when using ReLU as an activation function, which we make use in the following section to roughly estimate the size of the neural network.

For the last layer (output layer), a sigmoid function is used:

$$h(x) = \frac{1}{1 + \exp^{-x}},$$

in order to attain a value that can be interpreted as a probability.

### 3.1.1 Architecture

In this work we trained a fully connected, multi-layer Perceptron (MLP). We follow the results in [15], which establish lower bounds for the number of required non-zero weights and layers necessary for a MLP using ReLU activation functions used, to approximate the classification maps $f : [-1/2, 1/2]^d \to \mathcal{Y}$:

$$f = \sum_{k \leq N} c_k \mathcal{X}_{K_k},$$

where $f$ is a piecewise smooth function with $N$ "pieces". Here each $\mathcal{X}_{K_k}$ denotes an indicator function:

$$\mathcal{X}_{K_k}(x) = \begin{cases} 1, & \text{if } x \in K_k \\ 0, & \text{if } x \notin K_k \end{cases} \qquad (x,t) \in [0,1] \times \mathbb{R}^+, \tag{5}$$

for a compact set $K_k \subset [-1/2, 1/2]^d$ with $\partial K_k \in \mathcal{C}^\beta$, the set of $\beta$ times differentiable curves, for all $k \leq N$. Namely, $\beta$ controls the regularity of the boundary of the set $K_k$, $d$ denotes the dimension of the feature map.

Theorem 4.2 [15] provides the following lower bound for the minimal number of nonzero weights that a neural network $R_\rho(\Phi_\varepsilon^f)$ needs to have in order to approximate any function belonging to a function class $\mathcal{C} \in L^2([-1/2, 1/2]^d)$, up to an $L^2$ error of at most $\varepsilon$:

$$\#Weights \geq C \cdot \varepsilon^{-2(d-1)/\beta} / \log_2 \left( \frac{1}{\varepsilon} \right),$$

where $C = C(d, \beta, B, C_0)$ is a constant, $B, C_0 > 0$, $B$ is an upper bound on the supremum norm of $R_\rho(\Phi_\varepsilon^f)$ and $C_0$ is related to with the maximum bits needed to encode a weight of the network.

And Theorem 3.5 [15] provides approximation rates for piecewise constant functions. In particular, the number of required layers depends on the dimension $d$ and for the regularity parameter $\beta$:

$$\#Layers \geq c' \log_2(2 + \beta) \cdot (1 + \beta/d),$$

where $c' = c'(d, r, \beta, B) > 0$.

Making optimistic assumptions about the classification function and the desired approximation accuracy (in $L^2$ norm), we can bound, from below, the number of necessary neurons and layers as (non-zero) $\#Weights \approx 6.3 \times 10^7$ and $\#Layers \approx 2.4$, respectively, for $d = 11$, $\beta = 2$ and $\varepsilon = 0.10$. As these are lower bounds and $C$, $c'$ are unspecified, we fix the number of neurons to be at least $2^{28}$ which is distributed across the varying number of layers. In appendix A, the explicit architectures are detailed.

## 3.2 Integration of the method on a CFD code

We assume that exists a neural network which has been trained offline. Each layer can be fully characterized by the following information:

$$g_i(w_i, b_i, h_i(\cdot))$$

where $h_i$ denotes the activation function, $w_i$ the weights matrix and $b_i$ a bias vector.

Thus, there are two routines necessary to integrate a trained neural network with an existing code:

1. **Generation of features:** given the local solution $u$, generate the feature quantities $X(u)$ as specified on section 2.

2. **Prediction routine:** Given the features $X(u)$, it is necessary to evaluate the function $f$. As denoted previously, a layer $g_i$ is fully specified by parameters $w_i, b_i, h_i(\cdot)$. Once the neural network has been trained offline, the weights and biases can be loaded onto the CFD code. What remains to be implemented are the activation functions for the hidden layers and the activation function for the output layer in order to evaluate $f$ at some given input.

Then, the stabilisation algorithm reads:

**Data:** solution at cell $i$, $u_i$
**Result:** stabilised solution $\tilde{u_i}$
X = generateFeatures($u_i$);
label = predict(X);
**if** *label = 1* **then**
  $\tilde{u_i}$ = stabilisation($u_i$);
**else**
  $\tilde{u_i} = u_i$;
**end**

**Algorithm 1:** Limiting

For systems, we tried both limiting on conservative variables and primitive variables. Each variable is limited independently.

### 3.3 Measuring performance of model

In this work we use two sets of performance measures, namely:

1. Label prediction measures

2. $L^1$-norm of the solution

For the first set of measures, we can use typical metrics used in computational statistics and machine learning communities:

$$\frac{tp + tn}{tp + tn + fp + fn} \quad \text{(accuracy)}$$

$$\frac{tp}{tp + fn} \quad \text{(recall)}$$

$$\frac{tp}{tp + fp} \quad \text{(precision)}$$

where $tp$ is the number of correctly predicted positive labels, $tn$ the number of correctly predicted negative labels, $fp$ the number of incorrectly predicted positive labels and $fn$ the number of incorrectly predicted negative labels.

| Model | Description | Accuracy (%) | Recall (%) | Precision (%) |
|---|---|---|---|---|
| Random | randomly guessing | 50.00 | $\theta^a$ | 50.00 |
| Model 1 | 2 hidden layers (HL) | 77.96 | 69.17 | 24.13 |
| Model 2 | 3 HL | 95.52 | 80.66 | 84.19 |
| Model 3 | 4 HL | 95.15 | 79.26 | 82.67 |
| Model 4 | 5 HL | 95.14 | 81.71 | 81.01 |
| **Model 5** | 5 HL + weighted loss ($\omega = 5$) | 95.67 | 79.05 | 86.39 |

[a] $\theta$ denotes the estimate of the ratio between positive and negative labels, which could be measured empirically through the training data. Empirically, we have $\hat{\theta} = 0.1$

It is important to consider recall and precision because the distribution of labels can be imbalanced. If only accuracy is measured, it is possible to attain a high accuracy by solely predicting the majority label.

For the second set of measures, we consider the $L^1$-norm because it is the relevant measure for a CFD code and we can study the effect of this method on the empirical error of the numerical solution.

### 3.4 Implementation details

The neural network was set up and trained using Tensorflow with a python API. The numerical solver is written in Fortran, based on [16].

## 4 Numerical experiments

This section is split in two parts: in subsection 4.1, we show the performance of the trained neural network on an unseen validation set, and measure the performance in terms of the accuracy, recall and precision. The second part (subsections 4.2 and 4.3), we choose the model that performed the best and we integrate it with a CFD code. The model runs as a blackbox limiter (denoted as NN) and we compare its performance to the Minmod limiter and hierarchical high order limiter (denoted as HIO) through the $L^1$ error norm. We also compare the performance of this limiter with the *optimally tuned* TVD limiter, although we do not present convergence rates of the TVD limiter due to the dependency on the $M$ parameter. We perform some tests for the linear advection equation and Euler system of equations. The initial conditions are chosen as different from the ones used for the training.

### 4.1 Detection rate

We measure the performance of several models on a unseen validation set. We measure whether the models are able to predict the right labels which are assigned from hand tuned limiters.

On an unseen validation set, a subset of the dataset generated as describes in section 2.

Going forward, we select model 5 as it performs well and because the resulting size of the weights matrices per layer is significantly smaller than model 3. It is debatable whether the differences between models 3, 4 and 5 are statistically significant.

### 4.2 Linear Advection

Consider a linear advection equation with $a \in \mathbb{R}$:

$$\frac{\partial}{\partial t}u + a\frac{\partial}{\partial x}u = 0 \tag{6}$$

and periodic boundary conditions.

#### 4.2.1 Gaussian pulse

We consider the following initial condition:

$$u_0(x) = 1 + 3\exp(-100(x-0.5)^2) \quad (x,t) \in [0,1] \times \mathbb{R}^+ \tag{7}$$

with advection velocity $a = 1$.

The convergence is shown in tables 3 and 4 after one full crossing for orders 2 and 3. Furthermore, in figure 1 we show how the maxima is clipped using different methods for a grid size of $N = 40$. We note that the *Minmod limiter* clips the maximum value of the solution, whereas a well tuned *TVD limiter* does not. The *hierarchical high order limiter* (denoted as HIO) throughout this section behaves as Minmod for the second order case, but for third order it does not limit the solution. The learned limiter (denoted as NN) behaves similarly for both orders, but it is to note that it seems to outperform the HIO limiter for the second order case. It makes sense that the performance does not depend (as much as the HIO limiter) on the order of the method, as we train the algorithm with only nodal information.

| N | No Limiter | MinMod | HIO | NN |
|---|---|---|---|---|
| 10 | 0.197729 \| 0.4 | 0.389622 \| 0.2 | 0.389622 \| 0.2 | 0.353091 \| 0.2 |
| 20 | 0.055585 \| 1.8 | 0.142990 \| 1.4 | 0.142990 \| 1.4 | 0.107405 \| 1.7 |
| 40 | 0.008065 \| 2.3 | 0.038609 \| 1.7 | 0.038609 \| 1.7 | 0.017078 \| 2.2 |
| 100 | 0.001181 \| 2.3 | 0.005105 \| 1.9 | 0.005105 \| 1.9 | 0.002031 \| 2.3 |

**Table 3**: $L^1$ error for one crossing of the Gaussian pulse 7 using different limiters for order 2.

| N | No Limiter | MinMod | HIO | NN |
|---|---|---|---|---|
| 10 | 0.010765 \| 1.7 | 0.392397 \| 0.2 | 0.224807 \| 0.3 | 0.354754 \| 0.2 |
| 20 | 0.003666 \| 2.8 | 0.143413 \| 1.5 | 0.007157 \| 5.0 | 0.107731 \| 1.7 |
| 40 | 0.000192 \| 2.2 | 0.038610 \| 1.7 | 0.002575 \| 3.2 | 0.017128 \| 2.2 |
| 100 | 0.000078 \| 2.6 | 0.004917 \| 1.9 | 0.000781 \| 2.3 | 0.001750 \| 2.3 |

**Table 4**: $L^1$ error for one crossing of the Gaussian pulse 7 using different limiters for order 3.

#### 4.2.2 Smooth and square hat

Next, we consider the following initial conditions, which contain a smooth Gaussian pulse and a hat function:

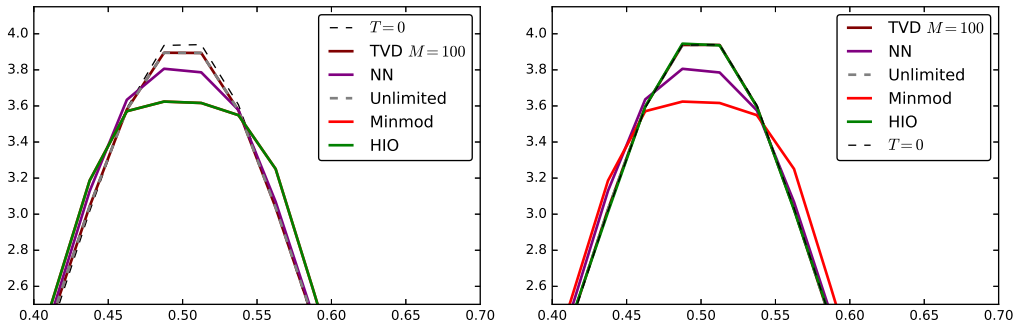


**Figure 1**: Maxima clipping of the Gaussian pulse 7 after one full crossing for approximation order of 2 (left) and 3 (right) for grid size $N = 40$.

$$u_0(x) = \begin{cases} 2, & |x - 0.7| \leq 0.1 \\ 1 + \exp\left(-\frac{(x-0.25)^2}{2\times 0.05^2}\right), & \text{otherwise} \end{cases} \qquad (x,t) \in [0,1] \times \mathbb{R}^+, \tag{8}$$

again with advection velocity $a = 1$.

| N | No Limiter | MinMod | HIO | NN |
|---|---|---|---|---|
| 10 | 0.164609 \| 0.4 | 0.333485 \| 0.2 | 0.333485 \| 0.2 | 0.297200 \| 0.3 |
| 20 | 0.100032 \| 0.7 | 0.170726 \| 1.0 | 0.170726 \| 1.0 | 0.129082 \| 1.2 |
| 40 | 0.053035 \| 0.8 | 0.062677 \| 1.2 | 0.062677 \| 1.2 | 0.065400 \| 1.1 |
| 80 | 0.029132 \| 0.8 | 0.028890 \| 1.2 | 0.028890 \| 1.2 | 0.032303 \| 1.1 |
| 100 | 0.024246 \| 0.8 | 0.023000 \| 1.2 | 0.023000 \| 1.2 | 0.026439 \| 1.0 |

**Table 5**: $L^1$ error for one crossing of the Gaussian and hat pulses 8 using different limiters for order 2.

| N | No Limiter | MinMod | HIO | NN |
|---|---|---|---|---|
| 10 | 0.054616 \| 0.6 | 0.339167 \| 0.2 | 0.307901 \| 0.3 | 0.302808 \| 0.3 |
| 20 | 0.032699 \| 0.7 | 0.172127 \| 1.0 | 0.075238 \| 2.0 | 0.130308 \| 1.2 |
| 40 | 0.022798 \| 0.6 | 0.063178 \| 1.2 | 0.033082 \| 1.6 | 0.065783 \| 1.1 |
| 80 | 0.013853 \| 0.6 | 0.029148 \| 1.2 | 0.017352 \| 1.4 | 0.032443 \| 1.1 |
| 100 | 0.011725 \| 0.7 | 0.023202 \| 1.2 | 0.014379 \| 1.3 | 0.026401 \| 1.0 |

**Table 6**: $L^1$ error for one crossing of the Gaussian and hat pulses 8 using different limiters for order 3.

### 4.3 Euler equation

Now we consider the 1-dimensional Euler equations, which describe the behavior of an inviscid flow. This system of equations describe the evolution of a density $\rho$, a velocity $u$, a pressure $p$ and total energy $E$.

$$\frac{\partial}{\partial t}\rho + \frac{\partial}{\partial x}(\rho u) = 0 \tag{9}$$

$$\frac{\partial}{\partial t}(\rho u) + \frac{\partial}{\partial x}(\rho u^2 + p) = 0 \tag{10}$$

$$\frac{\partial}{\partial t}E + \frac{\partial}{\partial x}((E + p)u) = 0. \tag{11}$$

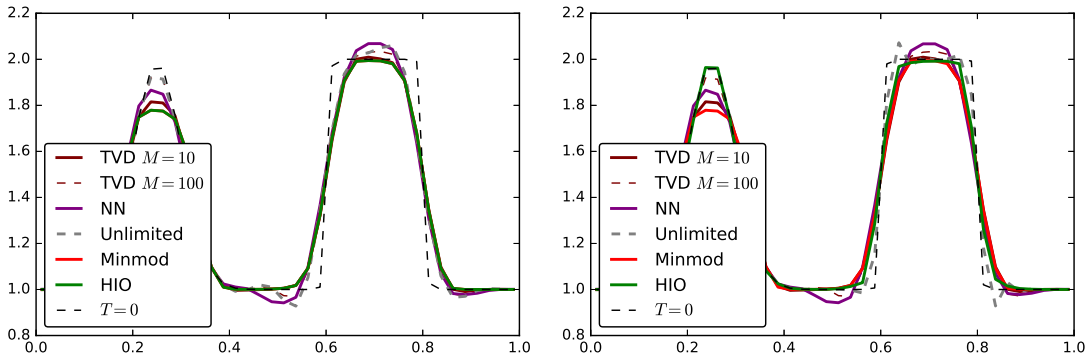


**Figure 2**: Maxima clipping of the Gaussian and hat pulses 8 after one full crossing for approximation order of 2 (left) and 3 (right) for grid size $N = 40$.

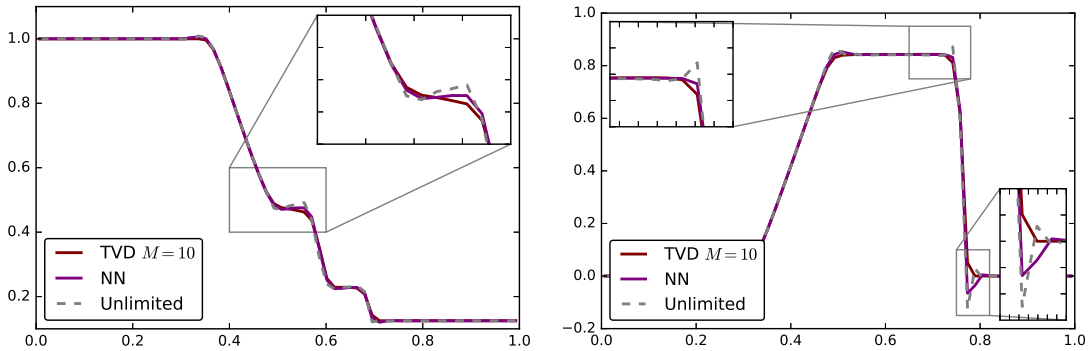


**Figure 3**: Density and velocity fields for the Sod shock tube 12 at $T = 0.1$ for grid size $N = 64$ and approximation order 2.

The system is closed equation of state for an ideal gas:

$$\rho e = p/(\gamma - 1),$$

where $e = E - \frac{1}{2}\rho u^2$ is the internal energy.

### 4.3.1 Sod shock tube

We consider the standard Sod shock tube test, given by the initial conditions:

$$(\rho, v, p)(x, 0) = \begin{cases} (1.0, 0.0, 1.0) & 0.0 < x \leq 0.5 \\ (0.125, 0.0, 0.1) & 0.5 < x < 1.0 \end{cases} \qquad (x, t) \in [0, 1] \times [0.0, 0.1], \qquad (12)$$

and $\gamma = 1.4$ and gradient free boundary conditions.

In figure 3 we show the comparison between different limiters at $T = 0.1$. We show the performance of a tuned TVD limiter and the NN limiter. We note that for the density field (left panel in figure 4.3.1), the solution produced by the NN limiter seems oscillation free and sharper than the solution by the TVD limiter, but for the velocity field (right panel in figure 4.3.1) some oscillations were not corrected enough, and clearly the behaviour of the TVD limiter is more desirable. This could be due to the fact that the shock detection is done over the conserved variables. This could be improved by performing the shock detection over the characteristic variables.

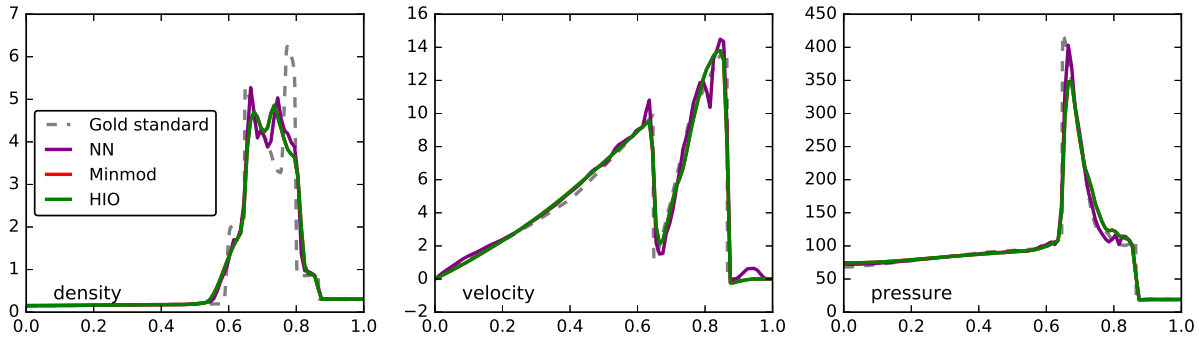


**Figure 4**: Density, velocity and pressure fields of the blast wave interaction 13 at $T = 0.038$ for grid size $N = 100$ and approximation order 2.

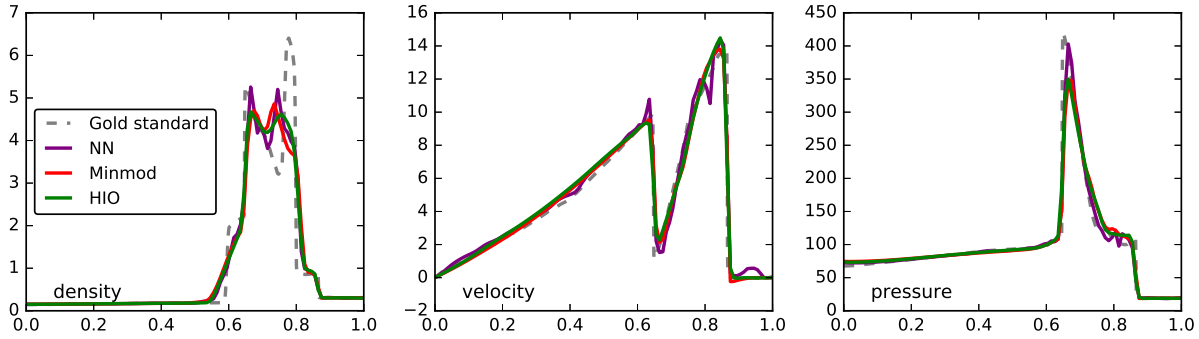


**Figure 5**: Density, velocity and pressure fields of the Blast Wave interaction at $T = 0.038$ for grid size $N = 100$ and approximation order 3.

### 4.3.2 Blast wave

Next we consider the interacting blast waves test, given by the initial conditions:

$$(\rho, v, p)(x, 0) = \begin{cases} (1.0, 0.0, 1000.0) & 0.0 < x \leq 0.1 \\ (1.0, 0.0, 0.01) & 0.1 < x \leq 0.9 \\ (1.0, 0.0, 100.0) & 0.9 < x < 1.0 \end{cases} \qquad (x, t) \in [0, 1] \times [0.0, 0.038] \qquad (13)$$

and $\gamma = 1.4$ and reflexive boundary conditions.

In figures 4 and 5 we show the comparison between different limiters at $T = 0.038$ for different orders. The unlimited solution is not shown because the code crashes due to the pressure becoming negative shortly after the start of the simulation for orders higher than 1. The dashed line denotes a high resolution solution, run with $N = 1000$, third order with the HIO limiter. We can note that the NN limiter is not as good at suppressing oscillations as Minmod and the HIO limiter, but stabilises the solution enough to finish the run. Furthermore, we note that the peak is better preserved, which means that it looks like the limiting is less strong than Minmod and the HIO limiter.

## 5 Conclusion

The purpose of this work was primarily to demonstrate the potential of using learning algorithms in CFD codes. In particular, we show the different stages necessary to train a blackbox limiter that can be integrated in different codes. To this end, we described how

to construct a dataset for the training phase, how to train a multi-layer Perceptron that detects shocks and how to integrate the trained model with a 1-dimensional discontinuous Galerkin code to serve as a limiter. Then, we performed the usual validation of the limiter in the context of scalar and systems of equations.

An idea that we tested in this paper was training a black box algorithm that mimics a limiter designed for a particular numerical scheme. In particular, we used the cells flagged by the hierarchical high order limiter [7] for discontinuous Galerkin method, which relies on the fact that the high order coefficients of the Legendre basis functions serve as proxies to the high order derivatives of the solution to perform the limiting, and we learn the mapping which using only nodal values of the solution as described in table 2. This is hopefully a step towards the development of algorithms which could be numerical scheme independent.

We showed empirically that in principle it's possible to learn a *parameter free* limiter (after the training phase) from data. While the performance for the seen functions is relatively good (as observed in 4.1), the on-the-fly performance could be improved with the careful design of the loss function during the training phase (for example, to include information on the maximum preserving principle). For the smooth case, we observed that this limiter was by far less diffuse than the *minmod* limiter. Certain oscillations were corrected but there were other oscillations which were not stabilized enough. When applied to a system of equations, we observed that this method still produced sensible results, slightly better than the unlimited version.

It is to note that the training set used was extremely reduced - we only used data from solving the linear advection equation with different initial conditions and varying mesh sizes. Even so, the direct application of this model to unseen initial conditions, to the Euler system and to differing grid sizes was somewhat successful. In a realistic setting, one would use a much more complete dataset.

Of course one must mention caveats in this type of work. For once, there is overhead when using the neural network limiter, as one can observe in table 7, where the average wall-clock time per timestep using with different limiters is shown. This can be improved significantly by using an efficient matrix multiplication routine to evaluate the prediction for each cell. A more concerning problem is the lack of theoretical guarantees for this type of black-box limiter. For example, additional work must be done to build limiters which are maximum principle preserving. These properties can be added by designing an appropriate loss function. In addition, work must be done for the extension to multi-dimensional problems.

| **Solver** | **N = 8** | **N = 16** | **N = 32** |
|---|---|---|---|
| No limiter | $8.15 \times 10^{-5}$ | $8.61 \times 10^{-5}$ | $1.29 \times 10^{-4}$ |
| TVD limiter | $8.70 \times 10^{-5}$ | $1.17 \times 10^{-4}$ | $1.74 \times 10^{-4}$ |
| NN limiter | $8.22 \times 10^{-3}$ | $1.68 \times 10^{-2}$ | $3.20 \times 10^{-2}$ |

**Table 7**: Average time per iteration (s) for sine advection case

While this work is a proof-of-concept, it is our belief that these ideas can be applied to other problems which depend on certain local properties of the numerical solution, ultimately contributing towards Computational Fluid Dynamics (CFD) codes which are robust to different initial conditions and that require less parameter tuning to produce

readily usable results.

## A Appendix: MLP architectures

Here we detail the architectures tested in this work. For this work we found the lower bound ($2^{28}$) for the number of weights and fixed the activation functions to be ReLU (as detailed in section 3.1.1). What we observed was that this quantity was not producing very good models. Thus, we used ($2^{32}$) non-zero weights, which presupposes a constant $C$ of order 10. It is left to specify the distribution of the weights across the different layers. The number of weights and neurons is related by multiplying $d$ through the number of neurons per layers to get the total number of weights.

| Model | Layers | Neurons per layer | Description |
|---|---|---|---|
| Model 1 | 2 | 16384:16384[1] | — |
| Model 2 | 3 | 2048:1024:512 | — |
| Model 3 | 4 | 512:256:256:128 | — |
| Model 4 | 5 | 256:128:64:64:32 | — |
| Model 5 | 5 | 256:128:64:64:32 | with weighted loss $\omega = 5$. |

**Table 8**: Architectures

## B Appendix: Adam algorithm

The Adam algorithm [11] is used for stochastic optimisation. The algorithm updates exponential moving averages of the gradient ($m_t$) and the squared gradients ($v_t$) where the hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay of these moving averages. These moving averages are estimates of the first and second moment (uncentered variance) of the gradient. Because the quantities $m_0, v_0$ are initialised at 0, they are biased towards zero (in particular if the decaying rates $\beta_1$ and $\beta_2$ are small). Hence, there is an additional correction leading to the unbiased quantities $\hat{m}_t$ and $\hat{v}_t$. The algorithm is detailed below.

**Data:** $\alpha$: stepsize, $\beta_1$, $\beta_2 \in [0, 1]$: exponential decay rates for the moment estimates, $f(\theta)$: stochastic objective function with parameters $\theta$, $\theta_0$ : initial parameter vector
**Result:** $\theta_t$ (resulting parameters)
$m_0 = 0$ (initialise 1st moment vector);
$v_0 = 0$ (initialise 2nd moment vector);
$t = 0$ (initialise timestep);
**while** $\theta_t$ *not converged* **do**
  $t = t + 1$;
  $g_t = \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t stochastic objective at stepsize $t$);
  $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate);
  $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second moment estimate);
  $\hat{m}_t = m_t / (1 - \beta_1^t)$ (Biased corrected first moment estimate);
  $\hat{v}_t = v_t / (1 - \beta_2^t)$ (Biased corrected second raw moment estimate);
  $\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t / \sqrt{\hat{v}_t + \epsilon}$ (Update parameters);
**end**

**Algorithm 2:** Adam algorithm

## REFERENCES


[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[2] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[3] R. Rojas. Deepest Neural Networks. *ArXiv e-prints*, July 2017.

[4] R. Rojas. Networks of width one are universal classifiers. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 4, pages 3124–3127 vol.4, July 2003.

[5] Ilya Sutskever and Geoffrey E. Hinton. Deep, narrow sigmoid belief networks are universal approximators. *Neural Comput.*, 20(11):2629–2636, November 2008.

[6] Mohit Arora and Philip L. Roe. A well-behaved tvd limiter for high-resolution calculations of unsteady flow. *Journal of Computational Physics*, 132(1):3 – 11, 1997.

[7] L. Krivodonova. Limiters for high-order discontinuous Galerkin methods. *Journal of Computational Physics*, 226:879–896, September 2007.

[8] Github repository. `https://github.com/hanveiga/1d-dg-nn`.

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

[10] J. Snyman. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Applied Optimization. Springer, 2005.

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

[12] J. Feldman and R. Rojas. *Neural Networks: A Systematic Introduction*. Springer Berlin Heidelberg, 1996.

[13] T. Ryan Hoens and Nitesh V. Chawla. *Imbalanced Datasets: From Sampling to Classifiers*. John Wiley & Sons, Inc., 2013.

[14] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

[15] P. Petersen and F. Voigtlaender. Optimal approximation of piecewise smooth functions using deep ReLU neural networks. *ArXiv e-prints*, September 2017.

[16] Private correspondence. Romain Teyssier.