Interpolating PDE solutions using feedforward neural networks

Patrick Kidger

August 24, 2018

Abstract

By training a feedfoward neural network on example solutions to a particular partial differential equation, we find that it is able to interpolate values for solutions to the partial differential equation with dramatically better accuracy than traditional interpolation methods. In particular, it is able to accurately interpolate values on 'peaks' of functions (where the function's derivative is discontinuous), where traditional interpolation methods do very poorly. Our code is available at `https://github.com/patrick-kidger/machinelearninginterpolation`.

Contents

1 Introduction **1**

2 The PDE of choice **2**

3 As a machine learning problem **3**
3.1 Data Generation 3
3.2 Preprocessing 5

4 Results **6**
4.1 Predictions across peaks 6
4.2 Interpolating other functions 8

A Implementation Details **9**
A.1 Data Generation and Numerical Analysis 9
A.2 Network Hyperparameters 11

1 Introduction

We consider the problem in which the values of a continuous function are known on a grid of points, and we wish to interpolate to approximate the value of the function at in-between points. Most traditional interpolation schemes make use of only the known values of the function on the grid – and certainly, if this is all that is known about the function, then that is all that can be done; but that is rarely the case. In particular, it may be known the function solves a particular partial differential equation.

In principle, the partial differential equation could be analysed and the form of its solutions determined. The function's known values on the grid could then allow for determining exactly which solution the function represents, effectively performing 'perfect' interpolation. In practice this is often easier said than done, but the fact remains that the extra information is there. By training a feedforward neural network on example solutions to the partial differential equation, we find that it is able to dramatically outperform traditional interpolation methods in both speed and accuracy.

From a machine learning point of view, this is both an easy and a hard problem. It is easy in the sense that interpolation can be done simply by outputting values close to the inputs. (The difficulty in getting a neural network to learn the identity function notwithstanding [HZRS16, THR+18].) But it is hard in the sense that in order to be useful, the neural network must outperform traditional methods, which may already get very close to the true value.

2 The PDE of choice

Our PDE of choice is the Camassa–Holm equation

$$u_t + 2\kappa u_x - u_{xxt} + 3uu_x = 2u_x u_{xx} + uu_{xxx}$$

in dimensionless spacetime variables (t, x). First obtained by Fuchssteiner and Fokas [FF81], and later rediscovered by Camassa and Holm [CH93, CH94], it is an integrable and bi-Hamiltonian equation which may be used to describe breaking waves in shallow water. There is an extensive literature available on the equation; [CS02, QZ06, Joh02, McK04, BSS99] is a sample. We shall simply recap some important details.[1]

Much of the literature, and indeed our own work, focuses on the special (although unphysical) case $\kappa = 0$:

$$u_t - u_{xxt} + 3uu_x = 2u_x u_{xx} + uu_{xxx}.$$

In this case, the solution admits 'peakon' solutions of the form

$$u(t, x) = c\mathrm{e}^{-|x-ct|}$$

interpreted in a suitable weak sense [CLP12, CM99, CE98]. Furthermore, it also admits multipeakon solutions of the form

$$u(t, x) = \sum_{j=1}^{n} p_j(t)\mathrm{e}^{-|x-x_j(t)|}$$

where the p_j, x_j evolve according to nonlinear interactions, which may in fact be described algebraically [BSS99, BSS00].

Given general initial data, it is possible for the solution to 'break down' in finite time, by developing a discontinuity in its first spatial derivative. That is, it develops a peak; this may be understood as the breaking of waves. It is surprising that it is neither the smoothness nor size of the initial data that determines whether or not breakdown occurs. Instead, it is the shape of the initial data [McK04, CE98]. In general, as $t \to \infty$, one tends to expect that a solution will asymptotically develop into a (potentially infinite) train of peakons [ET13]. An example of a typical solution is shown in Figure 2.1

[1] Wikipedia also provides a good introduction: `https://en.wikipedia.org/wiki/Camassa-Holm_equation`; accessed 13 August 2018.

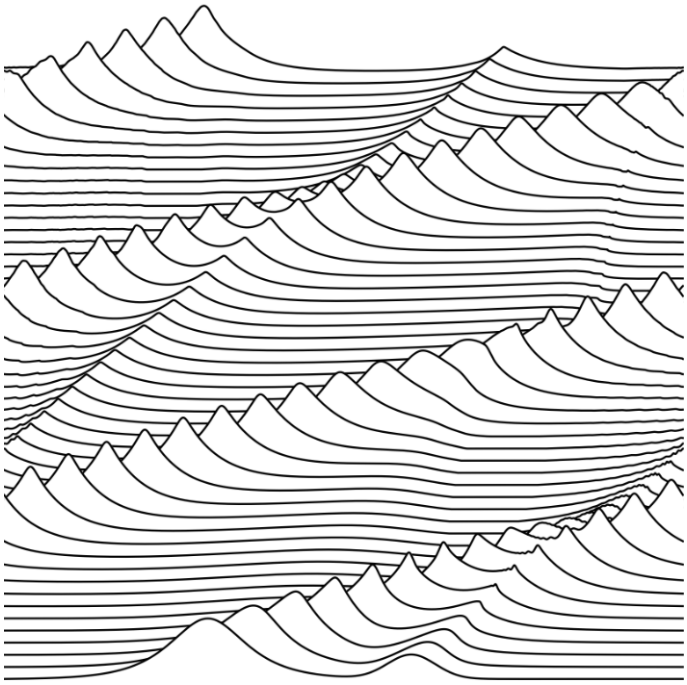


Figure 2.1: Evolution of the Camassa–Holm equation in a spatially-periodic domain $[0, 10] \times [0, 20]$ in time and space variables respectively, from initial condition $5\operatorname{sech}(x-6) + 2\exp(-(x-12)^2)$.

3 As a machine learning problem

The neural networks we use are just simple feedforward neural networks. The networks take the values of the solution on the grid as features, and produce the interpolated values of the solution as labels, either on a finer grid or at a particular point (in which case the requested point is a feature also). As such, we train two different sets of networks, corresponding to 'fine grid predictions' and 'single point predictions' respectively. We refer to these as the grid and point prediction problems respectively. See Figures 3.1 and 3.2. The precise implementation details are given in Appendix A.

As a side note, observe that our problem could be thought of as an imputing problem; we are seeking to fill in missing data.

3.1 Data Generation

We are in a fortunate position when it comes to the data on which the network is trained: the data may be generated simply by solving our partial differential equation numerically, and so we may generate an arbitrary amount of it with reasonable speed. As such, we have no risk of overfitting, as we never have to reuse data. The obvious caveat is that numerical solutions tend to have a slight error from the numerical discretisation. To help with this, we complement the numerical solutions of the equation – started from some random initial condition – with the well-behaved and well-understood single-peakon and multipeakon solutions of the equation, which may be generated precisely from their algebraic descriptions. Any given batch of data contains a mixture of the different kinds of solutions.

General numerical solutions were computed according to the scheme laid out in [Cot],

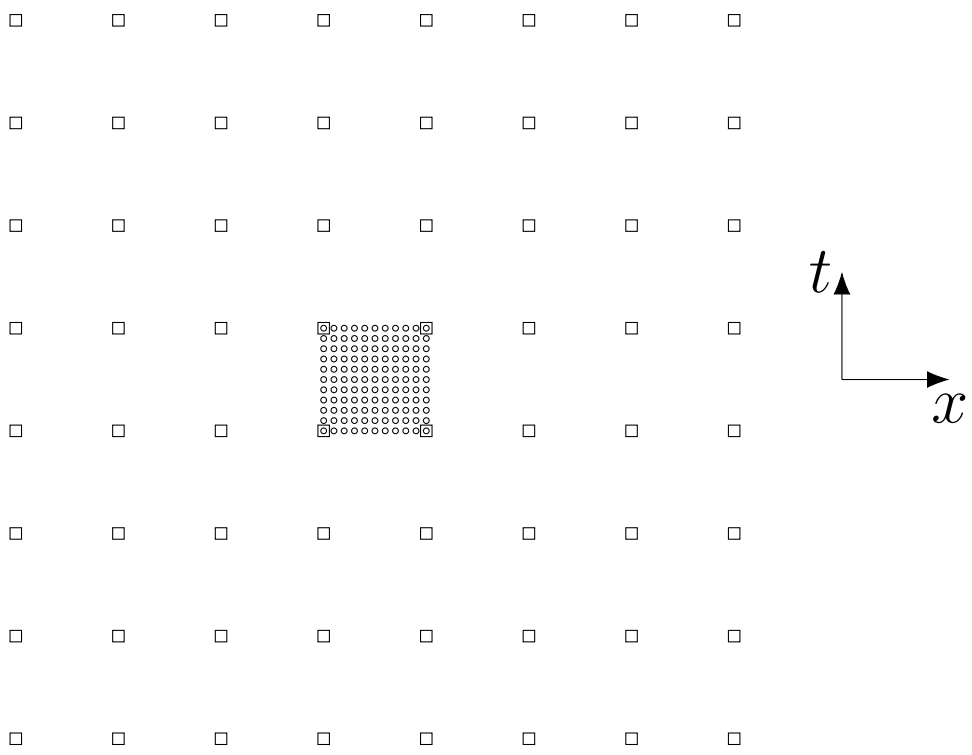


Figure 3.1: The network is given the values at the square points, and asked to predict the values at the central collection of round points.

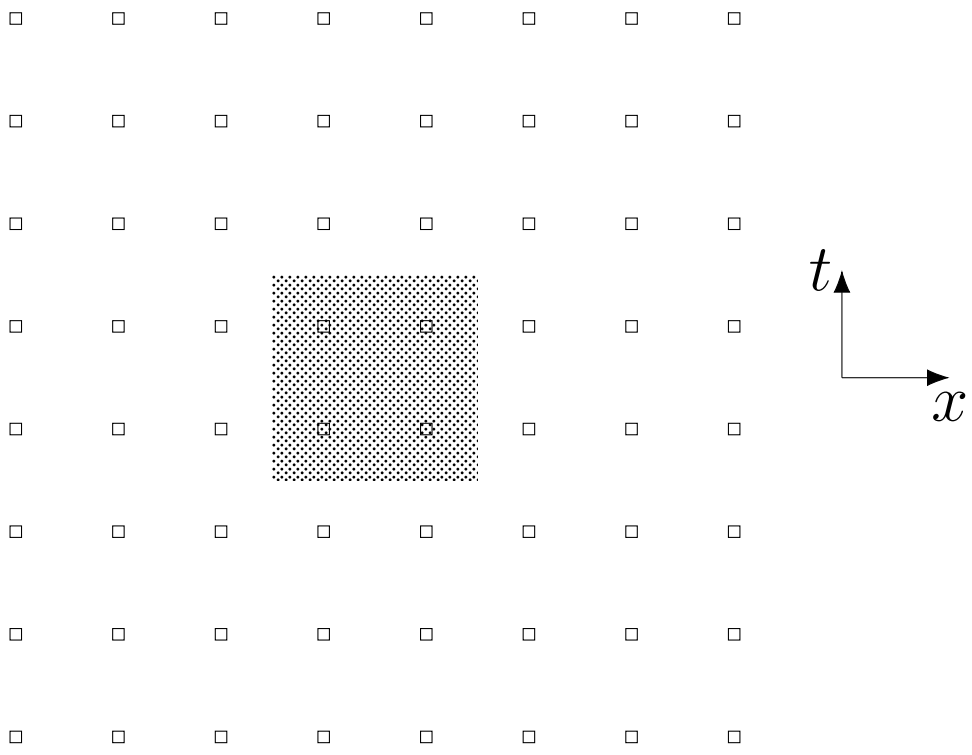


Figure 3.2: The network is given the values at the square points and a random location within the shaded region, and asked to predict the value at that location.

who cites [Mat10]. Generating a random single-peakon solution is trivial. Generating a random multipeakon solution is slightly harder, due to the nonlinear interactions of the peakons. Precise formulae for multipeakon solutions (under a different choice of scaling) are given in [BSS99, BSS00]. In the case of a two-peakon solution, these formulae may be inverted to give the solution

$$u(t,x) = p_1(t)\mathrm{e}^{-|x-x_1(t)|} + p_2(t)\mathrm{e}^{-|x-x_2(t)|}$$

where

$$\begin{aligned}
p_1(t) &= \frac{\lambda_1^2 a_1(t) + \lambda_2^2 a_2(t)}{\lambda_1\lambda_2(\lambda_1 a_1(t) + \lambda_2 a_2(t))},\\
p_2(t) &= \frac{a_1(t) + a_2(t)}{\lambda_1 a_1(t) + \lambda_2 a_2(t)},\\
x_1(t) &= \log\frac{(\lambda_1-\lambda_2)^2 a_1(t)a_2(t)}{\lambda_1^2 a_1(t) + \lambda_2^2 a_2(t)},\\
x_2(t) &= \log(a_1(t) + a_2(t)),
\end{aligned}$$

where

$$\begin{aligned}
a_1(t) &= A_1\mathrm{e}^{t/\lambda_1},\\
a_2(t) &= A_2\mathrm{e}^{t/\lambda_2},\\
A_1 &= \frac{(\lambda_2 P_2 - 1)\mathrm{e}^{X_2}}{P_2(\lambda_2-\lambda_1)},\\
A_2 &= \mathrm{e}^{X_2} - A_1,\\
\lambda_1 &= \frac{-(P_1+P_2)\mathrm{e}^{X_2} + \sqrt{(P_1+P_2)^2\mathrm{e}^{2X_2} + 4P_1P_2\mathrm{e}^{X_2}(\mathrm{e}^{X_1}-\mathrm{e}^{X_2})}}{2P_1P_2(\mathrm{e}^{X_1}-\mathrm{e}^{X_2})},\\
\lambda_2 &= \frac{-(P_1+P_2)\mathrm{e}^{X_2} - \sqrt{(P_1+P_2)^2\mathrm{e}^{2X_2} + 4P_1P_2\mathrm{e}^{X_2}(\mathrm{e}^{X_1}-\mathrm{e}^{X_2})}}{2P_1P_2(\mathrm{e}^{X_1}-\mathrm{e}^{X_2})},
\end{aligned}$$

where

$$\begin{aligned}
p_1(0) &= P_1,\\
p_2(0) &= P_2,\\
x_1(0) &= X_1,\\
x_2(0) &= X_2,
\end{aligned}$$

is the initial data, with $X_2 > X_1$.

Using three-peakon solutions and higher was not attempted, as they seemed unlikely to offer any benefit over the two-peakon case.

3.2 Preprocessing

As is usual, the features were normalised prior to being fed into the network. All of the values of the solution on the grid points must be collectively treated as one feature, as else they will lose their shape relative to the other values. In our case, we also applied the same transformation to the labels, as both tend to take similar values.

Regressor	Mean Squared Error	Maximum Absolute Error
Ensemble of the best 5 networks	3.53×10^{-3}	1.71
Network of 6 layers of 512 neurons	3.64×10^{-3}	1.70
Network of 2 layers of 64 neurons	7.75×10^{-3}	1.59
7th degree polynomial regression	4.59×10^{-1}	8.33
Bilinear interpolation	4.73×10^{-1}	7.44
9th degree polynomial regression	4.74×10^{-1}	8.71
5th degree polynomial regression	4.82×10^{-1}	5.63

Table 4.1: Losses for the point prediction problem; mean squared error is averaged over over 10 000 data points.

Both min-max normalisation and standardisation were used successfully (learning their values with momentum on a batch-by-batch basis[2]). However it is worth nothing that both of these depend on certain properties of the distribution that the solutions are drawn from, such as its mean, range, or standard deviation. We would not expect these properties to affect the problem of interpolation, which is more concerned with the values of a function locally, and the overall shape of the function that those values suggest.

As such, our choice of scaling is to normalise the data on an individual (not batch) basis, by scaling such that the values of solution on the corners of the central grid square have minimum 0 and maximum 1. Because the same scaling was also applied to the labels, then this means the neural network usually only has to predict values in a neighbourhood of $[0, 1]$, regardless of input.

4 Results

The neural networks give dramatically better results than either bilinear interpolation or polynomial regression. They perform better both on average, and in terms of the maximum error between prediction and reality. They are also, naturally, far quicker to evaluate than polynomial regression. Tables 4.1 and 4.2 show results for the best and worst neural networks, an ensemble of the best few neural networks, and bilinear interpolation and polynomial regression. See also Tables A.1 and A.2 for the complete results, including other sizes of neural network.

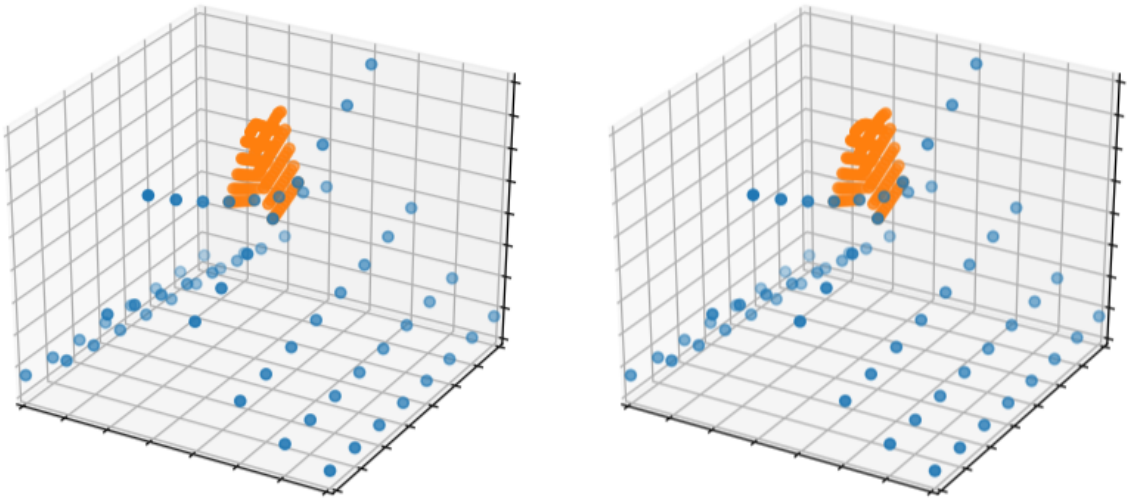
4.1 Predictions across peaks

Where the neural networks really shine is in making predictions across peaks, whether they are the sharp peaks of the single-peakon and multipeakon solutions, or the slightly curved nearly-peaks of the numerical solutions. Here, conventional interpolation algorithms will tend to round off the peak, whilst the neural network correctly predicts the peak. See Figure 4.3.

[2]So technically it's not quite min-max normalisation, as the learned range of the data is averaged across batches.

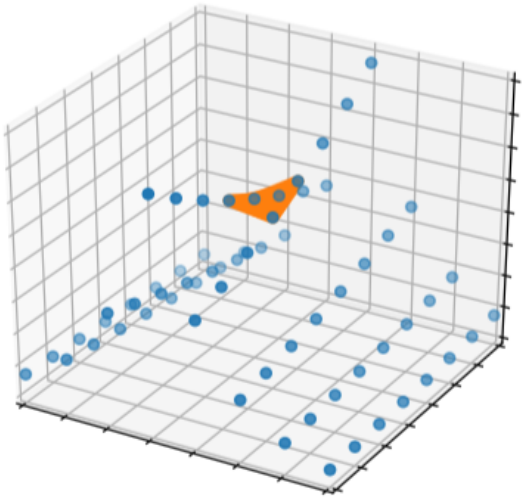
Regressor	Mean Squared Error	Maximum Absolute Error
Network of 8 layers of 256 neurons	1.04×10^{-3}	1.27
Ensemble of the best 5 networks	1.05×10^{-3}	1.22
Network of 2 layers of 64 neurons	1.52×10^{-3}	1.31
9th degree polynomial regression	4.95×10^{-2}	4.14
7th degree polynomial regression	5.36×10^{-2}	4.02
Bilinear interpolation	6.39×10^{-2}	5.10
5th degree polynomial regression	1.98×10^{-1}	5.62

Table 4.2: Losses for the grid prediction problem; mean squared error is averaged over 10 000 data points and all 121 output logits (corresponding to the points of the fine grid).

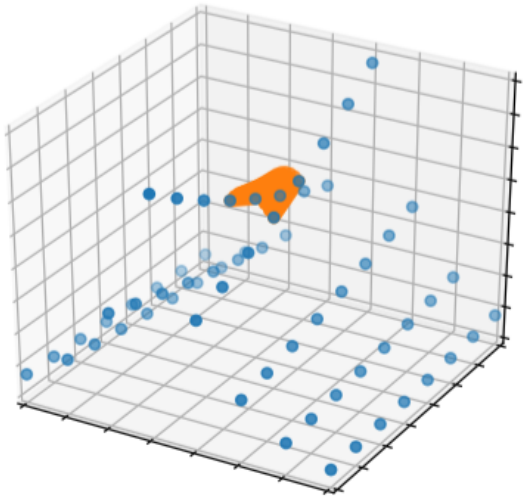


(a) True values

(b) Network of 8 layers of 256 neurons



(c) Bilinear interpolation



(d) 9th degree polynomial interpolation

Figure 4.3: Visual comparison of neural networks against traditional interpolation methods for interpolating over a peak in the grid prediction problem. Blue points show the grid of known input points. The central cluster of orange points shows the predictions.

4.2 Interpolating other functions

As seems usual for neural networks [THR+18], they fail to systematically generalise beyond data seen at training time. For example, asking the networks to interpolate functions of the form

$$(t, x) \mapsto \alpha \sin(\beta_1 t + \gamma_1) \sin(\beta_2 x + \gamma_2)$$

gives generally poor results.

Even peaked functions fail to get good predictions. Asking the networks to interpolate functions of the form[3]

$$(t, x) \mapsto \alpha \exp(-|x|)$$

gives equally poor results.

The failure to interpolate in general might seem disappointing, but it is actually quite encouraging; it serves to confirm our initial hypothesis that there is a reasonable amount of extra information encapsulated in the notion of a function solving a particular PDE, in particular information that the neural networks are able to exploit.

References

[ABH+15] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. Rognes, and G. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100):9–23, 2015.

[BSS99] R. Beals, D. Sattinger, and J. Szmigielski. Multipeakons and a theorem of Stieltjes. *Inverse Problems*, 15:L1–L4, 1999.

[BSS00] R. Beals, D. Sattinger, and J. Szmigielski. Multipeakons and the classical moment problem. *Adv. Math.*, 154(2):229–257, 2000.

[CE98] A. Constantin and J. Escher. Global existence and blow-up for a shallow water equation. *Ann. Scuola Norm. Sup. Pisa Cl. Sci. (4)*, 26:303–328, 1998.

[CH93] R. Camassa and D. Holm. An integrable shallow water equation with peaked solitons. *Phys. Rev. Lett.*, 71:1661–1664, 1993.

[CH94] R. Camassa and D. Holm. A new integrable shallow water equation. *Adv. Appl. Mech.*, 31:1–33, 1994.

[CLP12] A. Chertock, J.-G. Liu, and T. Pendleton. Convergence of a particle method and global weak solutions of a family of evolutionary PDEs. *SIAM J. Numer. Anal.*, 50(1):1–21, 2012.

[CM99] A. Constantin and H. McKean. A shallow water equation on the circle. *Comm. Pure Appl. Math.*, 52(8):949–982, 1999.

[Cot] C. Cotter. Camassa-Holm equation. `https://www.firedrakeproject.org/demos/camassaholm.py.html`. [Accessed 13 August 2018].

[3]Note the lack of t dependence.

[CS02] A. Constantin and W. Strauss. Stability of the Camassa-Holm solitons. *J. Nonlinear Sci. Vol.*, 12:415–422, 2002.

[ET13] J. Eckhardt and G. Teschl. On the isospectral problem of the dispersionless Camassa–Holm equation. *Adv. Math.*, 235:469–495, 2013.

[FF81] B. Fuchssteiner and A. Fokas. Symplectic structures, their Bäcklund transformation and hereditary symmetries. *Physica D*, 4:47–66, 1981.

[HZRS16] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks, 2016, `http://arxiv.org/abs/1603.05027`.

[Joh02] R. Johnson. Camassa–Holm, Korteweg–de Vries and related models for water waves. *J. Fluid Mech.*, 455:63–82, 2002.

[LM+12] A. Logg, G. Mardal, K.-A. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[Mat10] T. Matsuo. A Hamiltonian-conserving Galerkin scheme for the Camassa–Holm equation. *J. Comput. Appl. Math.*, 234(4):1258–1266, 2010.

[McK04] H. McKean. Breakdown of the Camassa-Holm equation. *Comm. Pure Appl. Math.*, 57(3):416–418, 2004.

[QZ06] Z. Qiao and G. Zhang. On peaked and smooth solitons for the Camassa-Holm equation. *Europhs. Lett.*, 74(5):657–663, 2006.

[SSAL16] W. Shang, K. Sohn, D. Almeida, and H. Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2217–2225. JMLR.org, 2016.

[THR+18] A. Trask, F. Hill, S. Reed, J. Rae, C. Dyer, and P. Blunsom. Neural arithmetic logic units, 2018, `http://arxiv.org/abs/1808.00508`.

A Implementation Details

A.1 Data Generation and Numerical Analysis

In our implementation, the network was told the values on an 8-by-8 spacetime grid, with points separated by a distance of 10^{-1}. In the case of the fine grid, the fine grid points were separated by a distance of 10^{-2}. Recall Figures 3.1 and 3.2.

General numerical solutions were computed using FEniCS [ABH+15, LM+12][4]. The implementation of this scheme was greatly aided by [Cot], who implements the scheme in a similar framework. Solutions to random initial conditions of the form

$$x \mapsto \sum_{i=1}^{n} \left[\alpha_i \operatorname{sech}(x - x_i) \sum_{j=0}^{m_i} \frac{1 + \sin(\beta_{ij} x + \gamma_{ij})}{2m_i} \right],$$

[4]See `https://fenicsproject.org`; accessed 13 August 2018. See also `https://fenicsproject.org/citing/` for more complete citations for the FEniCS project.

were computed in the spatially-periodic domain $[0, 10] \times [0, 20]$ in time and space variables respectively, where n, m_i, α_i, x_i, β_{ij}, γ_{ij} are randomly chosen constants. The distributions of these constants are (independent of each other):

$$\begin{aligned} n &\sim \text{Uniform}(\{2, 3\}) \\ m_i &\sim \text{Uniform}(\{2, 3, 4\}) \\ \alpha_i &\sim \text{Uniform}([3, 10]), \\ \beta_{ij} &\sim \text{Uniform}([-1.7, 1.7]), \\ \gamma_{ij} &\sim \text{Uniform}([-\pi, \pi]), \end{aligned}$$

whilst the x_i follow a more complicated distribution to ensure that they are reasonably well spaced (so that our solution is reasonably interesting nearly everywhere): the region is split up into n equally sized pieces, and each x_i placed uniformly at random within the central 70% of its piece.

Our numerical scheme has two further adaptions. First, we note that solutions should never go negative[5], but numerical errors often lead to discrepanices in that regard. A small smoothing around zero ensures that the solutions remain well-behaved. Explicitly, if u_h is a function representing the solution at some time, with expansion

$$u_h = \sum_{i=1}^{N} U_i \phi_i,$$

where $\{\phi_i \,|\, i \in \{1, \ldots, n\}\}$ is a basis of the finite element function space $V_h \ni u_h$, and $0 < \sigma \ll 1$ is fixed, then the U_i are corrected after calculation via the assignment

$$U_i \leftarrow \begin{cases} 0 & \text{if } U_i < -\sigma, \\ \frac{1}{4\sigma}(U_i + \sigma)^2 & \text{if } -\sigma \leqslant U_i < \sigma, \\ U_i & \text{if } U_i > \sigma. \end{cases}$$

Secondly, as we are in a periodic domain, it is natural that the values of the initial condition at the endpoints of the domain should be equal (else they tend to produce their own small ripple through the solution). This is resolved by adding on a small linear function to the intial condition; our choice of intial condition ensures that this correction need never be too large.

The size of the domain and the precise nature of the random initial condition were carefully chosen to ensure that the solution is usually 'interesting' at most points of the domain. We then choose random locations within the domain to place grids on, to generate feature/label pairs. As it is the local behaviour of the solution that is of interest, we obtain a speedup by sampling the same solution several times, at different locations. We found that sampling 1000 times produced good coverage of our domain without too much overlap.

Each element of a batch had a 50% chance of being a general numerical solution, a 40% chance of being a two-peakon solution, and a 10% chance of being a single-peakon solution.

[5]Actually, the author has been unable to find an explicit reference that this should be the case, despite several claims that the solution represents 'the height of the water's free surface above a flat bottom' [CH93, CH94], for example. The closest result we could find is Theorem 3.5 of [CE98], who treat the case that $u - u_{xx}$ is of a single sign.

Regressor	Loss	Maximum Error
Ensemble of the best 5 networks	3.53×10^{-3}	1.71
512×6 network	3.64×10^{-3}	1.70
512×10 network	3.91×10^{-3}	1.72
512×4 network	4.06×10^{-3}	1.76
512×8 network	4.09×10^{-3}	1.60
256×6 network	4.25×10^{-3}	1.75
256×4 network	4.46×10^{-3}	1.73
256×10 network	4.60×10^{-3}	1.65
256×8 network	4.68×10^{-3}	1.60
128×4 network	5.17×10^{-3}	1.61
128×6 network	5.17×10^{-3}	1.52
128×10 network	5.36×10^{-3}	1.48
128×8 network	5.74×10^{-3}	1.67
64×10 network	5.94×10^{-3}	1.58
64×4 network	5.96×10^{-3}	1.82
512×2 network	6.00×10^{-3}	1.21
64×6 network	6.19×10^{-3}	1.51
256×2 network	6.21×10^{-3}	1.40
64×8 network	6.38×10^{-3}	1.45
128×2 network	7.38×10^{-3}	1.62
64×2 network	7.75×10^{-3}	1.59
7th degree polynomial regression	4.59×10^{-1}	8.33
Bilinear interpolation	4.73×10^{-1}	7.44
9th degree polynomial regression	4.74×10^{-1}	8.71
5th degree polynomial regression	4.82×10^{-1}	5.63

Table A.1: Average loss and maximum error for the point prediction problem; loss is averaged over over 10 000 data points. $a \times b$ refers to a network of a layers of b neurons each.

A.2 Network Hyperparameters

Ad-hoc testing suggested Concatenated ReLU (CReLU) [SSAL16] as the best activation function. (Over ReLU, Leaky ReLU, ELU, or SELU.) The testing also showed that dropout had only adverse affects on the network's learning; this is perhaps not terribly surprising as there is no real risk of overfitting the training data; see Subsection 3.1. We note that dropout-style regularisation on the features might plausibly enhance a model's ability to make predictions for interpolating other functions, see Subsection 4.2, but this was not attempted. (In particular, dropping some of the input values from those grid points not forming part of the central square.)

A variety of models were trained, from 2 to 10 layers, and with 64 to 512 neurons per layer. The complete results are shown in Tables A.1 and A.2. For each problem, a single network of each size was trained for 15000 steps, with learning rates decreasing from 10^{-3} to 10^{-5} and batch sizes increasing from 32 to 512.

Regressor	Loss	Maximum Error
256×8 network	1.04×10^{-3}	1.27
Ensemble of the best 5 networks	1.05×10^{-3}	1.22
512×4 network	1.07×10^{-3}	1.12
256×6 network	1.09×10^{-3}	1.20
256×4 network	1.13×10^{-3}	1.38
128×6 network	1.13×10^{-3}	1.21
128×4 network	1.16×10^{-3}	1.11
128×8 network	1.16×10^{-3}	1.35
512×8 network	1.19×10^{-3}	1.28
128×10 network	1.20×10^{-3}	1.40
512×2 network	1.26×10^{-3}	1.18
256×2 network	1.27×10^{-3}	1.23
512×6 network	1.28×10^{-3}	1.28
64×6 network	1.29×10^{-3}	1.62
256×10 network	1.30×10^{-3}	1.12
512×10 network	1.34×10^{-3}	1.29
64×4 network	1.38×10^{-3}	1.30
128×2 network	1.38×10^{-3}	1.39
64×8 network	1.40×10^{-3}	1.60
64×10 network	1.49×10^{-3}	1.77
64×2 network	1.52×10^{-3}	1.31
9th degree polynomial regression	4.95×10^{-2}	4.14
7th degree polynomial regression	5.36×10^{-2}	4.02
Bilinear interpolation	6.39×10^{-2}	5.10
5th degree polynomial regression	1.98×10^{-1}	5.62

Table A.2: Average loss and maximum error for the grid prediction problem; loss is averaged over over 10 000 data points. Loss and error is per output logit. (There are 121, corresponding to the points of the fine grid.) $a \times b$ refers to a network of a layers of b neurons each.