

Package: nlmixr2nn (via r-universe)

October 13, 2026

Version 0.0.0.9000

Title Neural-Network ODEs for 'rxode2' and 'nlmixr2'

Description Embed neural networks inside ordinary differential equation right-hand sides for 'rxode2' and 'nlmixr2'. Networks are exposed as compiled 'rxode2' custom functions that supply the network value and its analytic input derivatives, allowing larger networks than the per-weight code-generation approach of 'pmxNODE'.

License GPL (>= 3)

URL <https://nlmixr2.github.io/nlmixr2nn/>,
<https://github.com/nlmixr2/nlmixr2nn>

BugReports <https://github.com/nlmixr2/nlmixr2nn/issues>

Encoding UTF-8

Imports rxode2, checkmate, stats, symengine

LinkingTo rxode2, Rcpp, nlmixr2est

SystemRequirements C++17. Optionally libtorch, via the 'torch' R package: when the 'torch' package and its libtorch binaries (torch::install_torch()) are present at build time, they are linked as the training backend. When they are not, a self-contained Adam/SGD optimizer is built instead, so the package installs and trains with no external library.

Suggests torch, nlmixr2, nlmixr2est, numDeriv, testthat (>= 3.0.0), knitr, rmarkdown, minqa, lbfgsb3c, n1qn1, ggplot2, callr, withr

VignetteBuilder knitr

NeedsCompilation yes

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Config/pak/sysreqs cmake libgmp3-dev make libicu-dev

Repository <https://nlmixr2.r-universe.dev>

Date/Publication 2026-09-20 21:07:42 UTC

RemoteUrl <https://github.com/nlmixr2/nlmixr2nn>
RemoteRef HEAD
RemoteSha d2b9e8bbfe5528fa25d2eebd8aad7a624e997ade

Contents

nn	2
nnControl	4
nnEval	8
nnWeights	9
plot.nlmixr2nnEval	9
Index	11

nn	<i>Neural-network term for an rxode2/nlmixr2 model</i>
----	--

Description

Use ‘nn(state, ..., nHidden=, act=)’ inside a model to insert a single-hidden-layer neural network of the state input(s). At parse time it is replaced by a compiled ‘nn<K>()’ call and its weights are added to the model as randomly-initialized population parameters.

Usage

```
nn(  
  ...,  
  nHidden = 5L,  
  act = c("softplus", "tanh", "relu", "gelu", "silu"),  
  init = c("ude", "torch", "normal"),  
  initSd = 0.1,  
  aug = 0L,  
  seed = NULL,  
  num = NULL,  
  iniDf = NULL  
)
```

Arguments

- ... one or more state/covariate inputs to the network (given positionally, e.g. ‘nn(central, t)’).
- nHidden hidden-layer width (default 5).
- act activation, one of "softplus" (default), "tanh", "relu", "gelu" or "silu". The default is smooth with a nonzero second derivative, which matters because the model’s Jacobian and the FOCEi sensitivities are both chained through it; "relu" has a kink and a zero second derivative.

init	weight initialization: "ude" (default) draws fan-in scaled weights with zero biases and a shrunk output layer, so an untrained network is a small perturbation of the mechanistic model; "torch" reproduces libtorch's 'nn::Linear' default; "normal" is a flat 'N(0, initSd)'.
initSd	scale of the initialization: the output-layer standard deviation for 'init = "ude"', and the standard deviation of every weight for 'init = "normal"'.
aug	number of AUGMENTED states to add (0-3, default 0) – the ANODE construction of Dupont et al. (2019). **Not available yet: any value above 0 is an error.** Augmenting from inside 'nn()' would renumber the model's compartments (the augmented state takes a 'd/dt()' slot next to the 'nn()' call, so 'cmt' in the data stops meaning what it means in the model), and a UDF cannot see the model's states to prevent that. Doing it at model assembly instead is the fix. Write the augmentation by hand meanwhile – see 'vignette("nlmixr2nn-node")'. 'aug = k' creates 'k' extra compartments 'rx_nnaug<id>_1..k', starts them at zero, gives each its own learned derivative, and passes all of them to this network as additional inputs. It is exactly what you would get by writing <pre> ““ d/dt(a1) <- nn(centr, a1, a2) d/dt(a2) <- nn(centr, a1, a2) g <- nn(centr, a1, a2) ““ </pre> by hand, and exists because the thing it fixes is invisible: a learned flow in one dimension cannot cross itself, so 'd/dt(centr) <- -f(nn(centr))' is topologically constrained no matter how well it is fitted or how much capacity it has. Extra dimensions remove the constraint. The cost is real and worth weighing. The augmented solve carries one variational state per (state x weight), and 'aug = k' multiplies BOTH: it adds 'k' states and 'k' more networks. Start at 1. With 'init = "ude"' (the default) every augmented derivative starts near zero, so the augmented states stay near zero and contribute nothing until training finds a use for them – the zero-augmentation start ANODE prescribes. (The network itself is still a fresh draw over the wider input vector, so this is not the same model as 'aug = 0'.) Total inputs ('...' plus 'aug') may not exceed 4. Each augmented state RELAXES rather than integrates: 'd/dt(a) = f(x, a) - a'. The textbook drift, 'd/dt(a) = f(x, a)', is self-exciting here – 'a' re-enters its own derivative through the network – and blows the solver up on a real dosing horizon. The '- a' gives it a stable fixed point at 'f', at the price of a fixed memory timescale of one model time unit.
seed	optional integer pinning THIS 'nn()' call's initial weights regardless of the ambient seed. Normally unnecessary – use [rxode2::rxSetSeed()]. Every network is drawn at 'seed + <network id>', so the networks of an 'aug =' call differ from each other rather than all drawing the same weights.
num	network occurrence number; queried via 'rxUdfUiNum()' if 'NULL'.
iniDf	initial-estimate data.frame; queried via 'rxUdfUiIniDf()' if 'NULL'.

Details

Inter-individual variability (a "deepNLME"/DeepPumas-style latent random effect) is best added by passing a per-subject latent eta as an INPUT to the population network, e.g. 'g <- nn(central,

eta.latent)’ with ‘eta.latent ~ omega’ in the ‘ini’ block. Because the eta is then an ordinary (visible) nlmixr2 random effect, its inner FOCEi sensitivity ‘d(g)/d(eta)’ is the network’s analytic input derivative – exact, no finite differences – and it is far more identifiable than a random effect on every weight.

The network’s initial weights are drawn WHEN THE MODEL IS PARSED, using rxode2’s threefry generator, and are carried on the model itself. So [rxode2::rxSetSeed()] makes a network reproducible across sessions – and, because threefry is the generator whose stream is defined under multiple threads, reproducible in the same places a parallel solve is. A freshly parsed model is immediately solvable: no torch module and no setup call.

Drawing weights disturbs nothing. Every draw runs inside [rxode2::rxWithSeed()], which saves and restores both the R stream and the rxode2 seed, so adding a network never shifts the draws elsewhere in a script. A bare ‘set.seed()’ still pins a model – it is the fallback used when ‘rxSetSeed()’ has not been called – but ‘rxSetSeed()’ is the one to reach for.

Value

a list consumed by [rxode2::rxUdfUi()] (‘replace’, ‘before’).

References

Dupont E, Doucet A, Teh YW (2019). "Augmented neural ODEs." *Advances in Neural Information Processing Systems* 32. See ‘vignette("nlmixr2nn-node")’ for what else from that literature does and does not carry over.

nnControl

Neural-network training schedule for ‘nlmixr2nn’

Description

Passed as the ‘nn=’ argument to [nlmixr2est::nlmixr2()] to train a model that contains an [nn()] term. It carries only the network training knobs – the inner NLME estimator and its control are the ordinary ‘est’/‘control’ arguments of ‘nlmixr2()’.

Usage

```
nnControl(
  mode = NULL,
  rounds = NULL,
  tol = NULL,
  wSteps = NULL,
  outerPerRound = NULL,
  lr = NULL,
  warmSteps = NULL,
  warmStart = NULL,
  warmPopIters = NULL,
  cotangent = NULL,
  optimizer = NULL,
```

```

    seed = NULL,
    l2 = NULL,
    smooth = NULL,
    kinetic = NULL
)

```

Arguments

mode	"joint" or "iter".
rounds	maximum number of rounds.
tol	convergence tolerance on the relative round-to-round change (weights for "iter"; weights and objective for "joint"). 0 always runs 'rounds'.
wSteps	weight-optimizer steps per round.
outerPerRound	("joint" only) inner outer-iterations per round – the partial step that makes it joint rather than a full re-fit.
lr	weight-optimizer learning rate.
warmSteps	optional naive-pooled (eta=0) warm-up steps before the loop (default 0).
warmStart	population weight pre-fit before the loop – the "nlm bridge". Runs a fixed-effects (eta = 0) fit of the weights with the named nlm-family optimizer, seeding the joint fit with a robust population weight vector (bridging the fixed-effects and DeepPumas approaches). Since nlm is simply an optimizer, the choice is "lbfgsb3c" (default), "nlminb", "nlm", "optim" (BFGS), "n1qn1" (gradient-based, using the analytic sensitivity gradient) or "bobyqa", "newuoa", "uobyqa" (derivative-free); "none" disables it. "pop" is an alias for "nlminb". The pre-fit optimizes only the weights and never uses a random effect, so any of these population-only optimizers applies. It is SKIPPED automatically when the model already carries trained weights (e.g. a fit passed back in) – those are the warm start. 'warmPopIters' caps its iterations.
warmPopIters	iteration cap for the 'warmStart' pre-fit (default 3). Kept deliberately light: a few iterations escape a bad random initialization while leaving the per-subject variation for the random effect. A longer population pre-fit can over-fit – the network then absorbs IIV that should go to the random effect (weakening eta recovery) and can push an SAEM inner fit into a poorly conditioned region – so raise it only when you want a stronger fixed-effects starting point (e.g. a population/UDE model with no random effect).
cotangent	source of the endpoint score dLL/df used to form the weight gradient. "gaussian" is the closed-form additive/proportional Gaussian cotangent. "dist" is the endpoint distribution's own derivative, for a count endpoint such as 'pois()' or 'binom()', where 'f' is the distribution's parameter rather than a prediction. "exact" is the per-observation cotangent captured from the inner fit's likelihood contribution hook, which is correct for any NORMAL residual model including transform-both-sides (lognormal, Box-Cox, logit) – best paired with 'wSteps = 1', since the captured cotangent is at the round's starting weights. "exact" does NOT apply to a count endpoint: there the hook reports $dLL/df = 1$, because 'f' is the log-density itself. Left unset, the source is chosen from the endpoint, which is what you want.

optimizer	weight optimizer, "adam" or "sgd". Supplied by libtorch when this build linked it and by the package's own implementation otherwise; given the same starting weights the two take the same steps.
seed	optional integer seed for weight initialization (ignored when the model already carries trained weights, which are used as the start).
l2	strength of an L2 penalty on the network's weights – the same idea as ridge regression, and what machine learning calls weight decay. The name is the L2 norm: the penalty is the SUM OF SQUARED weights, so a weight costs more the further it is from zero, and the fit prefers the flattest network consistent with the data. 'l2' is how much that preference is worth relative to fitting the observations; '0' expresses no preference at all. Concretely it adds $l2 * \sum(W_{eff}^2)$ over the weight MATRICES to the objective the weight step minimizes. Biases stay free, so the network can still move its output level without paying for it. 'Weff' is the effective weight – the first layer is penalized on $scale * W1$, because input scaling is folded into 'W1' (see 'nnScale.R') and raw 'W1' therefore means something different for a covariate of magnitude 500 than for one of magnitude 1. A latent eta's input column is exempt too: the eta reaches the model only through the network, so shrinking those weights would shrink the random effect itself. The value is a FRACTION, not an absolute amount: $l2 = 0.05$ means the weight term starts at 5 same thing across endpoints, estimators and data sizes. Defaults to '0' (off). This is measured, not cautious: L2 shrinks toward the zero function, and here the network IS the model, so any value large enough to suppress structure the data does not support also attenuates structure it does. Reach for it when a fit is visibly overfitting – '0.05' is where shrinkage becomes substantial (on a test fixture, weight norm 11.5 -> 3.9 with the unpenalized objective improving) – and check what the network learned with [nnEval()], because real effects shrink alongside invented ones.
smooth	strength of a curvature penalty – a roughness penalty in the spline sense, asking the network to be a SMOOTH function of its inputs rather than a small one. Where 'l2' pulls the network toward flat, 'smooth' pulls it toward gently curved, which is usually what you want from a learned covariate relationship: a shape, not a straight line and not an interpolation of noise. Concretely it adds $smooth * \sum(C^2)$ where $C = f(x+h) - 2*f(x) + f(x-h)$ is the second difference of the network along each input's marginal curve, that input swept over its observed range with the others held at their center. Penalizing the SECOND derivative rather than the first leaves monotone slopes free and charges only for the wiggles. Inputs whose range is not knowable – an eta, or a compound expression such as $nn(central/Vc)$ – are held fixed rather than swept. Like 'l2' it is a fraction of the objective and defaults to '0'; the two are normalized separately, because at equal lambda the raw curvature sum is orders of magnitude smaller than the raw weight sum and the mix would otherwise be arbitrary. Both penalties shape the OPTIMIZATION only. The reported 'objf' (and hence AIC/BIC) stays the unpenalized -2 log-likelihood, so a regularized network model remains directly comparable to an analytic-covariate one.

kinetic kinetic-energy (optimal-transport) penalty, as a FRACTION of the objective – though in a weaker sense than ‘l2’ and ‘smooth’. ‘kinetic = 0.05’ charges 5 than starting at exactly 5 term side is not. That is deliberate. ‘init = "ude"' starts the network near the zero function on purpose, so normalizing this term against its own starting value divides by something four orders of magnitude too small and the penalty then dominates as soon as the network’s output reaches its real size (measured: a penalty of 566 against an objective of 76). The normalizer is frozen once, from the first objective the fit produces, so the term is inert for that evaluation and live afterwards.

The term itself is ‘mean(f^2)’, where ‘f’ is the network’s output evaluated at the JOINTLY REALIZED input rows of a trial solve – the trajectory the model actually produces over the real dosing and time grid, not a synthetic grid. Where ‘smooth’ asks whether the learned function is wiggly in the abstract, this asks how hard the network is pushing where the solution really goes, which is a statement about the dynamics the ODE solver has to integrate.

It is the kinetic-energy / optimal-transport regularizer of the neural-ODE literature (Finlay et al. 2020; Worsham and Kalita 2025 measure it as the single largest improvement among the levers they compare, and one that made training FASTER because the learned dynamics became easier to integrate). That second effect is amplified here: the augmented solve carries one variational state per (state x weight), so every solver step a smoother right-hand side saves is multiplied by the whole sensitivity block.

OFF by default (‘0’). It changes the dynamics being solved rather than only steering the optimizer, and a default that quietly flattens a learned rate would be a default that quietly changes the model. Reach for it when the augmented solve is slow, or when the learned term looks stiffer than the data justifies, and check the result with [nnEval()]. Inputs the trial solve does not carry – an eta, or a compound expression such as ‘nn(central/Vc)’ – are held at their center rather than being paired with rows they never occurred with.

Details

Two modes: * “joint” (default) – co-optimizes the population parameters and the network weights in one interleaved loop (a few warm-started partial inner iterations + a weight-optimizer step each round), the DeepPumas-style approach; * “iter” – runs a full inner fit each round with the weights fixed, then a weight update (simpler solve/update/solve).

Value

an object of class “nnControl”.

Author(s)

Matthew L. Fidler

References

Finlay C, Jacobsen J-H, Nurbekyan L, Oberman AM (2020). "How to train your neural ODE: the world of Jacobian and kinetic regularization." *Proceedings of the 37th International Conference

on Machine Learning*, 3154-3164.

Worsham JM, Kalita JK (2025). "A guide to neural ordinary differential equations: machine learning for data-driven digital engineering." *Digital Engineering*, 100060. Reference implementation: <<https://github.com/joeworsh/Neural-ODE-Guide>>

The loop stops when the weights (and, for "joint", the objective) stop changing between rounds ('tol'), or after 'rounds'.

nnEval	<i>Evaluate a model's neural network at chosen inputs</i>
--------	---

Description

Evaluates the network embedded by [nn()] directly, at the weights the model currently carries – the trained weights of a fit, or the initial values drawn when the model was parsed. Use it to see the function the network represents.

Usage

nnEval(object, ..., net = 0L)

Arguments

- | | |
|--------|--|
| object | an 'rxUi' model or an 'nlmixr2' fit containing an [nn()] term. Named 'object' rather than 'x' deliberately: the inputs are supplied by name through '...', and a network input called 'x' is entirely ordinary – it would otherwise be captured by the first formal instead. |
| ... | the network inputs, given as named vectors (e.g. 'central = seq(0, 10, 0.1)'). Names are for labelling only; arguments are matched to the network's inputs positionally. |
| net | which network to evaluate, for a model containing more than one 'nn()' term (the id shown in the model text, counting from 0). |

Details

The inputs are supplied by name, in the order they appear in the 'nn()' call. Vectors of unequal length are recycled against each other, so a one-input network takes a single sequence and a two-input network can be swept over a grid built with [expand.grid()].

Value

a 'data.frame' of class '"nlmixr2nnEval"' with one column per input and a 'value' column holding the network output.

Author(s)

Matthew L. Fidler

Examples

```
mod <- function() {
  model({
    d/dt(central) <- -nn(central, nHidden = 4L)
  })
}
set.seed(1)
ui <- rxode2::rxode2(mod)
head(nnEval(ui, central = seq(0, 10, length.out = 5)))
```

nnWeights

*Network weights carried by a model or fit***Description**

Returns the weight vector a model currently uses: the values trained by a fit when it has been fitted, otherwise the initial values drawn when the model was parsed.

Usage

```
nnWeights(x)
```

Arguments

`x` an ‘rxUi’ model or an ‘nlmixr2’ fit built with [nn()].

Value

a named numeric vector of weights, or ‘NULL’ if the model has none.

Author(s)

Matthew L. Fidler

plot.nlmixr2nnEval

*Plot a network’s learned function***Description**

Plot a network’s learned function

Usage

```
## S3 method for class 'nlmixr2nnEval'
plot(x, true = NULL, ...)
```

Arguments

<code>x</code>	an <code>'nnEval()'</code> result.
<code>true</code>	optional reference to overlay: a function of the first input, or a numeric vector aligned with the rows of <code>'x'</code> . Use it to compare a learned term against the mechanism that generated the data.
<code>...</code>	passed to the underlying plotting call.

Value

a `'ggplot'` object when `ggplot2` is available, otherwise `'NULL'` invisibly (the plot is drawn).

Author(s)

Matthew L. Fidler

Index

`nn`, [2](#)
`nnControl`, [4](#)
`nnEval`, [8](#)
`nnWeights`, [9](#)
`plot.nlmixr2nnEval`, [9](#)