RobustNeuralNetworks.jl: a Package for Machine Learning and Data-Driven Control with Certified Robustness

Nicholas H. Barbara[1], Max Revay[1], Ruigang Wang[1], Jing Cheng[1], and Ian R. Manchester[1]

[1]University of Sydney, Australian Centre for Robotics

ABSTRACT

Neural networks are typically sensitive to small input perturbations, leading to unexpected or brittle behaviour. We present `RobustNeuralNetworks.jl`: a Julia package for neural network models that are constructed to naturally satisfy a set of user-defined robustness constraints. The package is based on the recently proposed Recurrent Equilibrium Network (REN) and Lipschitz-Bounded Deep Network (LBDN) model classes, and is designed to interface directly with Julia's most widely-used machine learning package, `Flux.jl`. We discuss the theory behind our model parameterization, give an overview of the package, and provide a tutorial demonstrating its use in image classification, reinforcement learning, and nonlinear state-observer design.

Keywords

Robustness, Machine Learning, Image Classification, Reinforcement Learning, State Estimation, Data-Driven Control

1. Introduction

Modern machine learning relies heavily on rapidly training and evaluating neural networks in problems ranging from image classification [5] to robotic control [18]. Most neural network architectures have no robustness certificates, and can be sensitive to adversarial attacks and other input perturbations [6]. Many that address this brittle behaviour rely on explicitly enforcing constraints during training to smooth or stabilize the network response [15, 9]. While effective on small-scale problems, these methods are computationally expensive, making them slow and difficult to scale up to complex real-world problems.

Recently, we proposed the *Recurrent Equilibrium Network* (REN) [16] and *Lipschitz-Bounded Deep Network* (LBDN) or *sandwich layer* [23] model classes as computationally efficient solutions to these problems. The REN architecture is flexible in that it includes all common neural network models, such as multi-layer-perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). The weights and biases in RENs are directly parameterized to *naturally satisfy* behavioural constraints chosen by the user. For example, we can build a REN with a given Lipschitz constant to ensure its output is quantifiably less sensitive to input perturbations. LBDNs are specializations of RENs with the specific feed-forward structure of deep neural networks like MLPs or CNNs and built-in guarantees on the Lipschitz bound.

The direct parameterization of RENs and LBDNs means that we can train models with standard, unconstrained optimization methods (such as stochastic gradient descent) while also guaranteeing their robustness. Achieving the "best of both worlds" in this way is the main advantage of the REN and LBDN model classes, and allows the user to freely train robust models for many common machine learning problems, as well as for more challenging real-world applications where safety is critical.

This papers presents `RobustNeuralNetworks.jl`: a package for neural network models that naturally satisfy robustness constraints. The package contains implementations of the REN and LBDN model classes introduced in [16] and [23], respectively, and relies heavily on key features of the Julia language [2] (such as multiple dispatch) for an efficient implementation of these models. The purpose of `RobustNeuralNetworks.jl` is to make our recent research in robust machine learning easily accessible to users in the scientific and machine learning communities. With this in mind, we have designed the package to interface directly with `Flux.jl` [8], Julia's most widely-used machine learning package, making it straightforward to incorporate our robust neural networks into existing Julia code.

The paper is structured as follows. Section 2 provides an overview of the `RobustNeuralNetworks.jl` package, including a brief introduction to the model classes (Sec. 2.1), their robustness certificates (Sec. 2.2), and their implementation (Sec. 2.3). Section 3 guides the reader through a tutorial with three examples to demonstrate the use of RENs and LBDNs in machine learning: image classification (Sec. 3.1), reinforcement learning (Sec. 3.2), and nonlinear state-observer design (Sec. 3.3). Section 4 offers some concluding remarks and future directions for robust machine learning with `RobustNeuralNetworks.jl`. For more detail on the theory behind RENs and LBDNs, and for examples comparing their performance to current state-of-the-art methods on a range of problems, we refer the reader to [16] and [23] (respectively).

2. Package overview

`RobustNeuralNetwork.jl` contains two classes of neural network models: RENs and LBDNs. This section gives a brief overview of the two model architectures and how they are parameterized to automatically satisfy robustness certificates. We also provide some background on the different types of robustness metrics used to construct the models.

2.1 What are RENs and LBDNs?

A *Lipschitz-Bounded Deep Network* (LBDN) is a (memoryless) deep neural network with a built-in upper-bound on its Lipschitz constant (Sec. *2.2.3*). Suppose the network has inputs $u \in \mathbb{R}^{n_u}$, outputs $y \in \mathbb{R}^{n_y}$, and hidden units $z_k \in \mathbb{R}^{n_k}$. The structure of an

LBDNs is a L-layer feed-forward network (like an MLP or CNN)

$$z_0 = x \tag{1}$$

$$z_{k+1} = \sigma(W_k z_k + b_k), \quad k = 0, \dots, L-1 \tag{2}$$

$$y = W_L z_L + b_L, \tag{3}$$

where the W_k, b_k are the layer weights and biases (respectively), and σ is a nonlinear activation function (e.g. $\tanh$, ReLU).

A *Recurrent Equilibrium Network* (REN) is a recurrent model (with memory) described by a linear dynamical system in feedback with a nonlinear activation function. Writing $x_t \in \mathbb{R}^{n_x}$ for the internal states of the system, a REN can be expressed mathematically as

$$\begin{bmatrix} x_{t+1} \\ v_t \\ y_t \end{bmatrix} = \overbrace{\left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{array}\right]}^{W} \begin{bmatrix} x_t \\ w_t \\ u_t \end{bmatrix} + \overbrace{\begin{bmatrix} b_x \\ b_v \\ b_y \end{bmatrix}}^{b}, \tag{4}$$

$$w_t = \sigma(v_t) := \begin{bmatrix} \sigma(v_t^1) & \sigma(v_t^2) & \cdots & \sigma(v_t^q) \end{bmatrix}^\top, \tag{5}$$

where $v_t, w_t \in \mathbb{R}^{n_v}$ are the inputs and outputs of the activation function σ. Graphically, this is equivalent to Figure 1, where the linear system G is given by Equation 4.

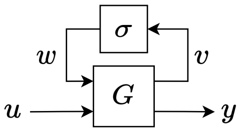

Fig. 1. Feedback structure of a recurrent equilibrium network.

See [16] and [23] for more on RENs and LBDNs, respectively.

REMARK 1. *[16] makes special mention of "acyclic" RENs, which have a lower-triangular D_{11} matrix. Acyclic RENs are significantly more efficient to evaluate than RENs with a dense D_{11} matrix, and performance is typically similar across a range of problems. All RENs in* `RobustNeuralNetworks.jl` *are therefore acyclic RENs.*

2.2 Robustness metrics and IQCs

All neural network models in `RobustNeuralNetworks.jl` are designed to satisfy a set of user-defined robustness constraints. The RENs can satisfy a range of robustness criteria, some relating to the internal dynamics of a model and others relating to its input-output map. LBDNs are more specialized and are specifically constructed to have a finite, user-tunable Lipschitz bound (Sec. *2.2.3*).

2.2.1 Contracting systems. Firstly, all of our RENs are contracting systems. This means that they exponentially "forget" their initial conditions. If the system starts at two different initial conditions but is given the same input sequence, the internal states will exponentially converge over time. Figure 2 shows an example of a contracting REN with one input and a single internal state, where two simulations of the system start with different initial conditions but are provided the same sinusoidal input. See [3] for a detailed introduction to contraction theory for dynamical systems.

2.2.2 Incremental IQCs. We define additional robustness criteria on the input-output map of RENs with incremental *integral quadratic constraints* (IQCs). Suppose we have a model $\mathcal{M}$ starting at two different initial conditions a, b with two different input signals u, v, and consider their corresponding output trajectories $y^a = \mathcal{M}_a(u)$ and $y^b = \mathcal{M}_b(v)$. The model $\mathcal{M}$ satisfies the IQC defined by matrices (Q, S, R) if

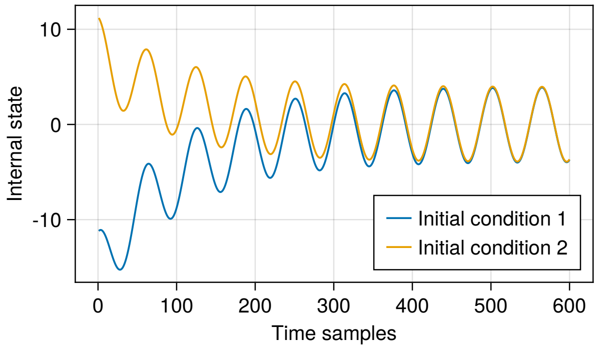

Fig. 2. Simulation of a contracting REN with a single internal state. The system is simulated from two different initial states with the same sinusoidal input. The contracting system exponentially forgets its initial condition.

$$\sum_{t=0}^{T} \begin{bmatrix} y_t^a - y_t^b \\ u_t - v_t \end{bmatrix}^\top \begin{bmatrix} Q & S^\top \\ S & R \end{bmatrix} \begin{bmatrix} y_t^a - y_t^b \\ u_t - v_t \end{bmatrix} \geq -d(a,b) \quad \forall T \tag{6}$$

for some function $d(a,b) \geq 0$ with $d(a,a) = 0$, where $0 \preceq Q \in \mathbb{R}^{n_y \times n_y}$, $S \in \mathbb{R}^{n_u \times n_y}$, $R = R^\top \in \mathbb{R}^{n_u \times n_u}$.

In general, the IQC matrices (Q, S, R) can be chosen (or optimized) to meet a range of performance criteria. The following special cases are worth noting.

2.2.3 Lipschitz bounds (smoothness). If $Q = -\frac{1}{\gamma} I$, $R = \gamma I$, $S = 0$ for some $\gamma \in \mathbb{R}$ with $\gamma > 0$, the model $\mathcal{M}$ satisfies a Lipschitz bound (incremental ℓ_2-gain bound) of γ defined by

$$\|\mathcal{M}_a(u) - \mathcal{M}_b(v)\|^2 \leq \gamma^2 \|u - v\|^2 \tag{7}$$

where $\|\cdot\|$ denotes the ℓ_2 norm. Qualitatively, the Lipschitz bound is a measure of the network's "smoothness". If γ is small, then small changes to the inputs u, v induce only small changes to the model output. If γ is large (or unbounded, as in the case of, e.g., MLPs and CNNs), then the model output can change significantly even with negligible changes to the inputs. This can make the model highly sensitive to noise, adversarial attacks, and other input disturbances.

As the name suggests, all LBDN models are constructed to have a user-tunable (or learnable) Lipschitz bound.

2.2.4 Incremental passivity. We have implemented two versions of incremental passivity. In each case, the network must have the same number of inputs and outputs.

(1) If $Q = 0, R = -2\nu I, S = I$ where $\nu \geq 0$, the model is incrementally passive (incrementally strictly input passive if $\nu > 0$). Mathematically, the following inequality holds.

$$\langle \mathcal{M}_a(u) - \mathcal{M}_b(v), u - v \rangle \geq \nu \|u - v\|^2 \tag{8}$$

(2) If $Q = -2\rho I, R = 0, S = I$ where $\rho > 0$, the model is incrementally strictly output passive. Mathematically, the following inequality holds.

$$\langle \mathcal{M}_a(u) - \mathcal{M}_b(v), u - v \rangle \geq \rho \|\mathcal{M}_a(u) - \mathcal{M}_b(v)\|^2 \tag{9}$$

For more details on IQCs and their use in RENs, please see [16].

2.3 Direct and explicit parameterizations

The key advantage of the models in `RobustNeuralNetworks.jl` is that they *naturally* satisfy the robustness constraints of Section 2.2 – i.e., robustness is guaranteed by construction. There is no need to impose additional (possibly computationally-expensive) constraints while training a REN or an LBDN. One can simply use unconstrained optimization methods like gradient descent and be sure that the final model will satisfy the robustness requirements.

We achieve this by constructing the weight matrices and bias vectors in our models to automatically satisfy specific linear matrix inequalities (see [16] for details). The learnable parameters of a REN or LBDN are a set of free, unconstrained variables $\theta \in \mathbb{R}^N$. When the set of learnable parameters is exactly $\mathbb{R}^N$ like this, we call the parameterization a *direct parameterization*. Equations 4 to 3 describe the *explicit parameterizations* of RENs and LBDNs: model structures that can be called and evaluated on data. For a REN, the explicit parameters are $\bar{\theta} := [W, b]$, and for an LBDN they are $\bar{\theta} := [W_0, b_0, \dots, W_L, b_L]$. The mapping $\theta \mapsto \bar{\theta}$ depends on the specific robustness constraints to be imposed on the explicit model.

2.3.1 Implementation. RENs are defined by two abstract types in `RobustNeuralNetworks.jl`. Subtypes of `AbstractRENParams` hold all the information required to directly parameterize a REN satisfying some robustness properties. For example, to initialize the direct parameters of a *contracting* REN with 1 input, 10 states, 20 neurons, 1 output, and a `relu` activation function, we use the following. The direct parameters θ are stored in `model_ps.direct`.

```
using Flux, RobustNeuralNetworks

T   = Float32
nu, nx, nv, ny = 1, 10, 20, 1
model_ps = ContractingRENParams{T}(
               nu, nx, nv, ny; nl=Flux.relu)

println(model_ps.direct) # Access direct params
```

Subtypes of `AbstractREN` represent RENs in their explicit form which can be evaluated on data. The conversion from direct to explicit parameters $\theta \mapsto \bar{\theta}$ is performed when the REN is constructed and the explicit parameters $\bar{\theta}$ are stored in `model.explicit`.

```
model = REN(model_ps)     # Create explicit model
println(model.explicit)   # Access explicit params
```

Figure 3 illustrates this architecture. We use a similar interface based on `AbstractLBDNParams` and `AbstractLBDN` for LBDNs.

2.3.2 Types of direct parameterizations. There are currently four REN parameterizations implemented in this package:

(1) `ContractingRENParams` parameterizes contracting RENs with a user-defined upper bound on the contraction rate.

(2) `LipschitzRENParams` parameterizes RENs with a user-defined (or learnable) Lipschitz bound $\gamma \in (0, \infty)$.

(3) `PassiveRENParams` parameterizes incrementally input passive RENs with user-tunable passivity parameter $\nu \geq 0$.

(4) `GeneralRENParams` parameterizes RENs satisfying some general behavioural constraints defined by an incremental IQC with parameters (Q,S,R).

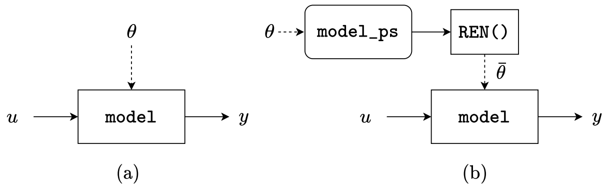

Fig. 3. Association of models and their parameters in (a) `Flux.jl` and (b) `RobustNeuralNetworks.jl`. In (a), model parameters θ are associated with the `model`. In (b), the direct parameters θ are associated with the parameterization `model_ps`, and are converted to explicit parameters $\bar{\theta}$ when the `model` is constructed for evaluation with `REN()`.

There is currently one LBDN parameterization implemented in `RobustNeuralNetworks.jl`:

(1) `DenseLBDNParams` parameterizes dense (fully-connected) LBDNs with a user-defined or learnable Lipschitz bound. A dense LBDN is effectively a Lipschitz-bounded MLP.

We intend to add `ConvolutionalLBDNParams` to parameterize the convolutional LBDNs in [23] in future iterations of the package.

2.3.3 Explicit model wrappers. When training a REN or LBDN, we learn and update the direct parameters θ and convert them to the explicit parameters $\bar{\theta}$ only for model evaluation. The main constructors for explicit models are `REN` and `LBDN`.

Users familiar with `Flux.jl` will be used to creating a model once and then training it on their data. The typical workflow is as follows.

```
using Flux

# Define a model and a loss function
model = Flux.Chain(
    Flux.Dense(1 => 10, Flux.relu),
    Flux.Dense(10 => 1, Flux.relu)
)

loss(model, x, y) = Flux.mse(model(x), y)

# Training data of 20 batches
T = Float32
xs, ys = rand(T,1,20), rand(T,1,20)
data = [(xs, ys)]

# Train the model for 50 epochs
opt_state = Flux.setup(Adam(0.01), model)
for _ in 1:50
    Flux.train!(loss, model, data, opt_state)
end
```

When training a model constructed from `REN` or `LBDN`, we need to back-propagate through the mapping from direct (learnable) parameters to the explicit model. We must therefore include the model construction as part of the loss function. If we do not, then the auto-differentiation engine has no knowledge of how the model parameters affect the loss, and will return zero gradients. Here is an example with an `LBDN`, where the `model` is defined by the direct parameterization stored in `model_ps`.

```
using Flux, RobustNeuralNetworks

# Define model parameterization and loss function
T = Float32
model_ps = DenseLBDNParams{T}(1, [10], 1; nl=relu)
```

```
function loss(model_ps, x, y)
    model = LBDN(model_ps)
    Flux.mse(model(x), y)
end

# Training data of 20 batches
xs, ys = rand(T,1,20), rand(T,1,20)
data = [(xs, ys)]

# Train the model for 50 epochs
opt_state = Flux.setup(Adam(0.01), model_ps)
for _ in 1:50
    Flux.train!(loss, model_ps, data, opt_state)
end
```

2.3.4 Separating parameters and models. For the sake of convenience, we have included the model wrappers `DiffREN` and `DiffLBDN` as alternatives to `REN` and `LBDN`, respectively. These wrappers compute the explicit parameters each time the model is called rather than just once when they are constructed. Any model created with these wrappers can therefore be used exactly the same way as a regular `Flux.jl` model, and there is no need for model construction in the loss function. One can simply replace the definition of the `Flux.Chain` model in the `Flux.jl` example with

```
model_ps = DenseLBDNParams{T}(1, [10], 1; nl=relu)
model = DiffLBDN(model_ps)
```

and train the LBDN just like any other `Flux.jl` model. We use these wrappers in Sections 3.1 and 3.3.

The trade-off in using `DiffREN` or `DiffLBDN` is computational efficiency in applications where a model is called many times before a training update (e.g., reinforcement learning). The main computational bottleneck in training a REN or LBDN is converting from the direct to explicit parameters (mapping $\theta \mapsto \bar{\theta}$). This process involves a matrix inverse where the number of matrix elements scales quadratically with the dimension of the model in a REN or the dimension of each layer in an LBDN (see [16, 23]). If a model is to be evaluated many times with the same direct parameters in between training updates, it is more efficient to compute the explicit parameters once, hold them fixed over many model calls, and only re-compute them once the direct parameters have been updated. This is exactly the purpose of keeping `model_ps` and `model` separate when using `REN` and `LBDN`. Note that we cannot store the direct and explicit parameters in the same `model` object since auto-differentiation in Julia does not support array mutation [7]. We therefore advise using `DiffREN` or `DiffLBDN` for convenience in applications where the model parameters are updated after just one model call (e.g., training an image classifier). The computational benefits of separating models from their parameterizations is explored numerically in Section 3.2.

3. Examples

This section guides the reader through a set of examples to demonstrate how to use `RobustNeuralnetworks.jl` for machine learning in Julia. We will consider three examples: image classification, reinforcement learning, and nonlinear state-observer design. These examples will provide further insight into the benefits of using robust models and the reasoning behind key design decisions made in the development of the package.

We use `relu` activation functions in all examples, but other choices of activation function (e.g: `tanh`) are equally valid. We note that any activation function used in a REN or LBDN must have a maximum slope of 1.0, as outlined in [16, 23]. For more examples with RENs and LBDNs, please see the package documentation[1].

3.1 Image classification

Our first example features an LBDN trained to classify the MNIST dataset [11]. We will use this example to demonstrate how training image classifiers with LBDNs makes them robust to noise (and adversarial attacks) thanks to the built-in Lipschitz bound. For a detailed investigation of the effect of Lipschitz bounds on classification robustness and reliability, please see [23].

3.1.1 Load the data. We begin by loading the training and test data. `MLDatasets.jl`[2] contains a number of common machine-learning datasets, including the MNIST dataset. To load the full dataset of 60,000 training images and 10,000 test images, one would run the following code.

```
using MLDatasets: MNIST

T = Float32
x_train, y_train = MNIST(T, split=:train)[:]
x_test,  y_test  = MNIST(T, split=:test)[:]
```

The feature matrices `x_train` and `x_test` are three-dimensional arrays where each 28×28 layer contains pixel data for a single handwritten number from 0 to 9 (e.g., see Fig. 4). The labels `y_train` and `y_test` are vectors containing the classification of each image as a number from 0 to 9. We convert each of these to a format better suited to training with `Flux.jl`.

```
using Flux

# Reshape features for model input
x_train = Flux.flatten(x_train)
x_test  = Flux.flatten(x_test)

# Encode categorical outputs and store
y_train = Flux.onehotbatch(y_train, 0:9)
y_test  = Flux.onehotbatch(y_test,  0:9)
data = [(x_train, y_train)]
```

Features are now stored in a $28^2 \times N$ `Matrix` where each column contains pixel data from a single image, and the labels have been converted to a $10 \times N$ `Flux.OneHotMatrix` where each column contains a 1 in the row corresponding to the image's classification (e.g., row 3 for an image showing the number 2) and a 0 otherwise.

3.1.2 Define a model. We can now construct an LBDN model to train on the MNIST dataset. The larger the model, the better the classification accuracy will be, at the cost of longer training times. The smaller the Lipschitz bound γ, the more robust the model will be to input perturbations (such as noise in the image). If γ is too small, however, it can restrict the model flexibility and limit the achievable performance [23]. For this example, we use a small network of two 64-neuron hidden layers and set a Lipschitz bound of $\gamma = 5.0$ just to demonstrate the method.

[1] https://acfr.github.io/RobustNeuralNetworks.jl/
[2] https://juliaml.github.io/MLDatasets.jl/

```
using RobustNeuralNetworks

# Model specification
nu = 28*28                # Inputs (size of image)
ny = 10                   # Outputs (classifications)
nh = fill(64,2)           # Hidden layers
γ  = 5.0f0                # Lipschitz bound 5.0

# Define parameters,create model
model_ps = DenseLBDNParams{T}(nu, nh, ny, γ)
model = Chain(DiffLBDN(model_ps), Flux.softmax)
```

The `model` consists of two parts. The first is a callable `DiffLBDN` model constructed from its direct parameterization, which is defined by an instance of `DenseLBDNParams` as per Section 2.3. The output is then converted to a probability distribution using a `softmax` layer. Note that all `AbstractLBDN` models can be combined with traditional neural network layers using `Flux.Chain`.

We could also construct the `model` as a chain of `SandwichFC` layers. Introduced in [23], the “sandiwch” layer is a dense layer with a guaranteed Lipschitz bound of 1.0. We have designed the user interface for `SandwichFC` similarly to that of `Flux.Dense`.

```
model = Chain(
    (x) -> (sqrt(γ) * x),
    SandwichFC(nu => nh[1], relu; T),
    SandwichFC(nh[1] => nh[2], relu; T),
    (x) -> (sqrt(γ) * x),
    SandwichFC(nh[2] => ny; output_layer=true, T),
    Flux.softmax
)
```

This model is equivalent to a dense LBDN constructed with `LBDN` or `DiffLBDN`. We have included it as a convenience for users familiar with layer-wise network construction in `Flux.jl`, and recommend using it interchangeably with `DiffLBDN`.

3.1.3 Define a loss function. A typical loss function for training on datasets with discrete labels is the cross entropy loss. We can use the `crossentropy` loss function shipped with `Flux.jl`.

```
loss(model,x,y) = Flux.crossentropy(model(x), y)
```

3.1.4 Train the model. We train the model over 600 epochs using two learning rates: `1e-3` for the first 300, and `1e-4` for the last 300. We use the `Adam` optimizer [10] and the default `Flux.train!` method for convenience. Note that the `Flux.train!` method updates the learnable parameters each time the model is evaluated on a batch of data, hence our choice of `DiffLBDN` as a model wrapper.

```
# Hyperparameters
epochs = 300
lrs = [1e-3,1e-4]

# Train with the Adam optimizer
opt_state = Flux.setup(Adam(lrs[1]), model)
for k in eachindex(lrs)
    for i in 1:epochs
        Flux.train!(loss, model, data, opt_state)
    end
    Flux.adjust!(opt_state, lrs[2])
end
```

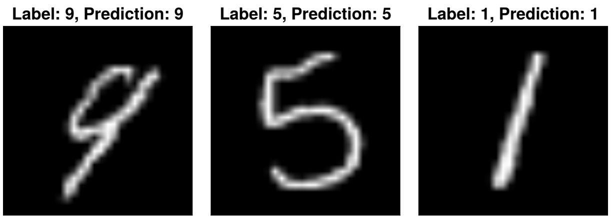

Fig. 4. Examples of classifications from the trained LBDN model on the MNIST dataset.

3.1.5 Evaluate the trained model. Our final model achieves training and test accuracies of approximately 98% and 97%, respectively, as shown in Table 1. We could improve this further by switching to a convolutional LBDN, as in [23]. Some examples of classifications given by the trained LBDN model are presented in Figure 4.

3.1.6 Investigate robustness. The main advantage of using an LBDN for image classification is its built-in robustness to noise (or attacks) added to the image. This robustness is a direct benefit of the Lipschitz bound. As outlined in the Section *2.2.3*, the Lipschitz bound effectively defines how “smooth” the network is: the smaller the Lipschitz bound, the less the network outputs will change as the inputs vary. For example, small amounts of noise added to the image will be less likely to change its classification. A detailed investigation into this effect is presented in [23].

We can demonstrate the robustness of LBDNs by comparing the model to a standard MLP built from `Flux.Dense` layers. We first create a `dense` network with the same layer structure as the LBDN.

```
# Initialisation functions
init = Flux.glorot_normal
initb(n) = Flux.glorot_normal(n)

# Build a dense model
dense = Chain(
    Dense(nu => nh[1], relu;
          init, bias=initb(nh[1])),
    Dense(nh[1] => nh[2], relu;
          init, bias=initb(nh[2])),
    Dense(nh[2] => ny; init, bias=initb(ny)),
    Flux.softmax
)
```

Training the `dense` model with the same training loop used for the LBDN model results in a model that achieves training and test accuracies of approximately 98% and 97%, respectively, as shown in Table 1.

Table 1. Training and test accuracy for the LBDN and Dense models on the MNIST dataset without perturbations.

Model structure	Training accuracy (%)	Test accuracy (%)
LBDN	98.2	97.2
Dense	97.6	96.6

As a simple test of robustness, we add uniformly-sampled random noise in the range $[-\epsilon, \epsilon]$ to the pixel data in the test dataset for a range of noise magnitudes $\epsilon \in [0, 200/255]$. We record the test accuracy for each perturbation size and store it for plotting.

```
using Statistics

# Get test accuracy as we add noise
uniform(x) = 2*rand(T, size(x)...) .- 1
compare(y, yh) =
    maximum(yh, dims=1) .== maximum(y.*yh, dims=1)
accuracy(model, x, y) = mean(compare(y, model(x)))

function noisy_test_error(model, ϵ)
    noisy_xtest = x_test .+ ϵ*uniform(x_test)
    accuracy(model, noisy_xtest,  y_test)*100
end

ϵs = T.(LinRange(0, 200, 10)) ./ 255
lbdn_error  = noisy_test_error.((model,), ϵs)
dense_error = noisy_test_error.((dense,), ϵs)
```

Plotting the results in Figure 5 very clearly shows that the `dense` network, which has no guarantees on its Lipschitz bound, quickly loses its accuracy as small amounts of noise are added to the image. In contrast, the LBDN `model` maintains its accuracy even when the (maximum) perturbation size is as much as 80% of the maximum pixel values. This is an illustration of why image classification is such a promising use-case for LBDN models. For a more detailed comparison of LBDN with state-of-the-art image classification methods, see [23].

3.2 Reinforcement learning

One of the original motivations for developing the model structures in `RobustNeuralNetworks.jl` was to guarantee stability and robustness in learning-based control. Recently, we have shown that with a controller architecture based on a nonlinear version of classical Youla-Kucera parameterization [25], one can learn over a space of stabilizing controllers for linear and nonlinear systems using standard reinforcement learning techniques, so long as the control policy is parameterized by a contracting, Lipschitz-bounded REN [24, 22, 1]. This is an exciting result for learning-based controllers in safety-critical systems, such as in robotics.

In this example, we will demonstrate how to train an LBDN controller with *reinforcement learning* (RL) for a simple nonlinear dynamical system. This controller will not have any stability guarantees. The purpose of this example is simply to showcase the steps required to set up RL experiments for more complex systems with RENs and LBDNs.

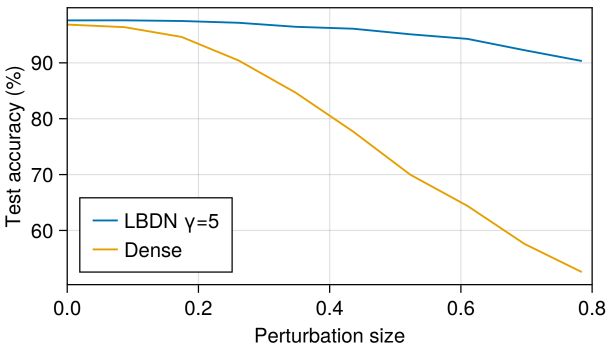

Fig. 5. Comparison of test accuracy on the MNIST dataset as a function of random perturbation magnitude ϵ. The LBDN model is significantly more robust than a standard `Dense` network.

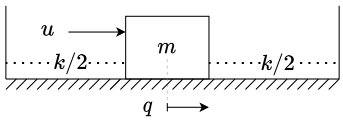

Fig. 6. Mechanical system to be controlled. A box sits in a tub of fluid, suspended between two springs, and can be pushed by a force u to different horizontal positions q.

3.2.1 *Overview.*

Consider the simple mechanical system shown in Figure 6: a box of mass m sits in a tub of fluid, held between the walls by two springs each with spring constant $k/2$. The box can be pushed with a force u. Its dynamics are

$$m\ddot{q} = u - kq - \mu\dot{q}|\dot{q}| \tag{10}$$

where μ is the viscous damping coefficient due to the box moving through the fluid, and $\dot{q}, \ddot{q}$ denote the velocity and acceleration of the box, respectively.

We can write this as a (nonlinear) state-space model with state $x = (q, \dot{q})^\top$, control input u, and dynamics

$$\dot{x} = f(x, u) := \begin{bmatrix} \dot{q} \\ (u - kq - \mu\dot{q}|\dot{q}|)/m \end{bmatrix}. \tag{11}$$

This is a continuous-time model of the dynamics. For our purposes, we need a discrete-time model. We can discretize the dynamics using a forward Euler approximation to get

$$x_{t+1} = f_d(x_t, u_t) := x_t + \Delta t \cdot f(x_t, u_t) \tag{12}$$

where Δt is the time-step. This approximation typically requires a small time-step for numerical stability, but is sufficient for our simple example. If physical accuracy was of concern, one could use a fourth (or higher) order Runge-Kutta scheme.

Our aim is to learn a controller $u = \mathcal{K}_\theta(x, q_{\text{ref}})$, defined by some learnable parameters θ, that can push the box to any goal position q_{ref} that we choose. Specifically, we want the box to:

(1) reach a (stationary) goal position q_{ref}

(2) within a time period T.

The force required to keep the box at a static equilibrium position q_{ref} is $u_{\text{ref}} = kq_{\text{ref}}$ from Equation 10. We can encode these objectives into a cost function J_θ and write our RL problem as

$$\min_\theta \mathbb{E}\left[J_\theta\right], \quad J_\theta = \sum_{t=0}^{T-1} c_1(\Delta q_t)^2 + c_2\dot{q}_t^2 + c_3(\Delta u_t)^2 \tag{13}$$

where $\Delta q_t = q_t - q_{\text{ref}}$, $\Delta u_t = u_t - u_{\text{ref}}$, c_1, c_2, c_3 are cost function weights, and the expectation is over different initial and goal positions of the box.

3.2.2 *Problem setup.*

We start by defining the properties of our system and translating the dynamics into Julia code. For this example, we consider a box of mass $m = 1$, spring constants $k = 5$, and a viscous damping coefficient $\mu = 0.5$. We will simulate the system over $T = 4$ s time horizons with a time-step of $\Delta t = 0.02$ s.

```
m = 1                           # Mass (kg)
k = 5                           # Spring constant (N/m)
μ = 0.5                         # Viscous damping (kg/m)
Tmax = 4                        # Simulation horizon (s)
dt = 0.02                       # Time step (s)
ts = 1:Int(Tmax/dt)             # Array of time indices
```

Now we can generate the training data. Suppose the box always starts at rest from the zero position, and the goal position can be anywhere in the range $q_{\text{ref}} \in [-1, 1]$. Our training data consists of a batch of 80 randomly-sampled goal positions and corresponding reference forces u_{ref}.

```
nx, nref, batches = 2, 1, 80
x0 = zeros(nx, batches)
qref = 2*rand(nref, batches) .- 1
uref = k*qref
```

It is good practice (and faster) to simulate all simulation batches at once, so we define our dynamics functions to operate on batches of states and controls. Each row corresponds to a different state or control, and each column corresponds to a simulation for a particular goal position.

```
f(x::Matrix,u::Matrix) = [x[2:2,:]; (u[1:1,:] -
    k*x[1:1,:] - μ*x[2:2,:]*abs.(x[2:2,:]))/m]
fd(x::Matrix,u::Matrix) = x + dt*f(x,u)
```

RL problems typically involve simulating the system over some time horizon and collecting rewards or costs at each time step. Control policies are trained using approximations of the cost gradient ∇J_θ, as it is often difficult (or impossible) to compute the exact gradient due to the complexity of dynamics simulators. We refer the reader to [20] for further details, and `ReinforcementLearning.jl` [21] for examples in Julia.

For this simple example, we can back-propagate directly through the dynamics function `fd(x,u)` rather than approximating ∇J_θ. The simulator below takes a batch of initial states, goal positions, and a controller `model` whose inputs are $[x; q_{\text{ref}}]$. It computes trajectories of states and controls $z = \{[x_0; u_0], \ldots, [x_{T-1}; u_{T-1}]\}$. To avoid the issue of unsupported array mutation when differentiating we use a `Zygote.Buffer` to iteratively store the outputs [7].

```
using Zygote: Buffer

function rollout(model, x0, qref)
    z = Buffer([zero([x0;qref])], length(ts))
    x = x0
    for t in ts
        u = model([x;qref])
        z[t] = vcat(x,u)
        x = fd(x,u)
    end
    return copy(z)
end
```

After computing these trajectories, we will need a function to evaluate the cost given some weightings c_1, c_2, c_3.

```
using Statistics

weights = [10,1,0.1]
function _cost(z, qref, uref)
    Δz = z .- [qref; zero(qref); uref]
    return mean(sum(weights .* Δz.^2; dims=1))
end
cost(z::AbstractVector, qref, uref) =
    mean(_cost.(z, (qref,), (uref,)))
```

3.2.3 *Define a model.*

We will train an LBDN controller with a Lipschitz bound of $\gamma = 20$. Its inputs are the state x_t and goal position q_{ref}, while its outputs are the control force u_t. We have chosen a model with two hidden layers each of 32 neurons just as an example. For examples of how Lipschitz bounds can be useful in learning robust controllers, see [1, 17, 19].

```
using RobustNeuralNetworks

T  = Float64
γ  = 20                   # Lipschitz bound
nu = nx + nref            # Inputs (x and reference)
ny = 1                    # Outputs (control action)
nh = fill(32, 2)          # Hidden layers
model_ps = DenseLBDNParams{T}(nu, nh, ny, γ)
```

3.2.4 *Define a loss function.*

In constructing a loss function for this problem, we refer to Section *2.3.3*. The `model_ps` contain all information required to define a dense LBDN model. However, `model_ps` is not a model that can be evaluated on data: it is a *model parameterization*, and contains the learnable parameters θ. To train an LBDN given some data, we construct the model within the loss function using the `LBDN` wrapper so that the mapping from direct to explicit parameters is captured during back-propagation. Our loss function therefore includes the following three components.

```
function loss(model_ps, x0, qref, uref)
    model = LBDN(model_ps)           # Model
    z = rollout(model, x0, qref)     # Simulation
    return cost(z, qref, uref)       # Cost
end
```

3.2.5 *Train the model.*

Having set up the RL problem, all that remains is to train the controller. The function below trains a model and keeps track of the training loss `tloss` (cost J_θ) for each simulation in our batch of 80. Training is performed with the `Adam` optimizer over 250 epochs with a learning rate of 10^{-3}.

```
using Flux

function train_box_ctrl!(
    model_ps, loss_func;
    epochs=250, lr=1e-3
)
    costs = Vector{Float64}()
    opt_state = Flux.setup(Adam(lr), model_ps)
    for k in 1:epochs

        tloss, dJ = Flux.withgradient(
            loss_func, model_ps, x0, qref, uref)
        Flux.update!(opt_state, model_ps, dJ[1])
        push!(costs, tloss)
    end
    return costs
end

costs = train_box_ctrl!(model_ps, loss)
```

3.2.6 *Evaluate the trained model.*

We may now verify the performance of the trained model on a new set of reference positions. In the code below, we generate 60 batches of test data. In each one, the box starts at the origin at rest, and is moved through the fluid to a different (random) goal position $q_{\text{ref}} \in [-1, 1]$. We plot the

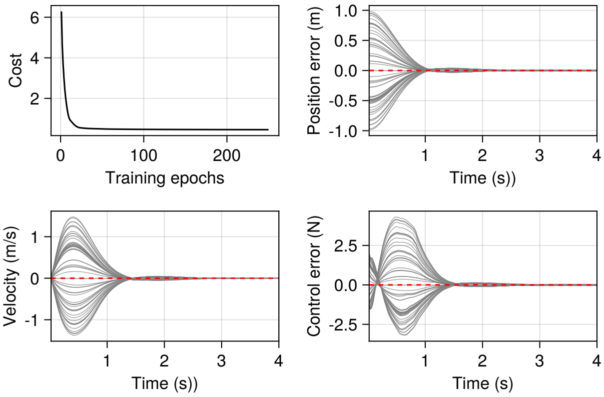

Fig. 7. Loss curve and simulation results from the LBDN RL policy controlling the box system in Figure 6. The LBDN policy can push the box to any desired location in the domain of interest. The position and controller errors are Δq and Δu from Equation 13, respectively.

states and controls alongside the loss curve from training in Figure 7. The box clearly moves to the required position within the time frame in all cases, experimentally verifying the performance of our controller.

```
model   = LBDN(model_ps)
x0_test = zeros(2,60)
qr_test = 2*rand(1, 60) .- 1
z_test  = rollout(model, x0_test, qr_test)
```

3.2.7 Advantages of separate parameters and models. As discussed in Section *2.3.4*, there is a trade-off between convenience and performance in `RobustNeuralNetworks.jl`. The `DiffLBDN` and `DiffREN` wrappers exist to allow users to train robust models in a `Flux.jl`-like manner. These wrappers convert a model parameterization to its explicit form each time they are called, hence the user does *not* have to re-construct the model in the loss function.

```
loss2(model, x0, qref, uref) =
    cost(rollout(model, x0, qref), qref, uref)
```

The cost is computation speed, particular in an RL context. Careful inspection of the `rollout()` function shows that the `model` is evaluated many times within the loss function before the learnable parameters are updated with `Flux.update!()`. As discussed in the Section *2.3.4*, the major computational bottleneck in training RENs and LBDNs is the conversion from learnable (direct) parameters to an explicit model. Constructing the model only when the parameters are updated therefore saves considerably on computation time, particularly for large models.

For example, suppose we train single-hidden-layer LBDNs with $n = 2, 2^2, \ldots, 2^9$ neurons over 100 epochs on our box RL problem, and log the time taken to train each model when using both `LBDN` and `DiffLBDN`.

```
function lbdn_compute_times(n; epochs=100)

    # Build model params and a model
    lbdn_ps = DenseLBDNParams{T}(nu, [n], ny, γ)
    diff_lbdn = DiffLBDN(deepcopy(lbdn_ps))

    # Time with LBDN vs DiffLBDN (respectively)
    t_lbdn = @elapsed (
        train_box_ctrl!(lbdn_ps, loss; epochs))
    t_diff_lbdn = @elapsed (
        train_box_ctrl!(diff_lbdn, loss2; epochs))
    return [t_lbdn, t_diff_lbdn]

end

# Evaluate computation time
# Run it once first for just-in-time compiler
ns = 2 .^ (1:9)
lbdn_compute_times(2; epochs=1)
comp_times = reduce(hcat, lbdn_compute_times.(ns))
```

The results are plotted in Figure 8. Even for a single-layer LBDN with $2^9 = 512$ neurons, it is clear that using `DiffLBDN` takes an order of magnitude longer to train than only constructing the `LBDN` model each time the `loss()` function is called. If we were training dynamic models with REN, the computational overhead of using `DiffREN` instead of `REN` would be even more extreme, since the conversion from direct to explicit parameters in a REN is typically more computationally expensive than for LBDNs. It is for this reason that we strongly recommend using the `LBDN` and `REN` wrappers if many evaluations of the model are required before `Flux.update!()` (or equivalent) is called, as in RL.

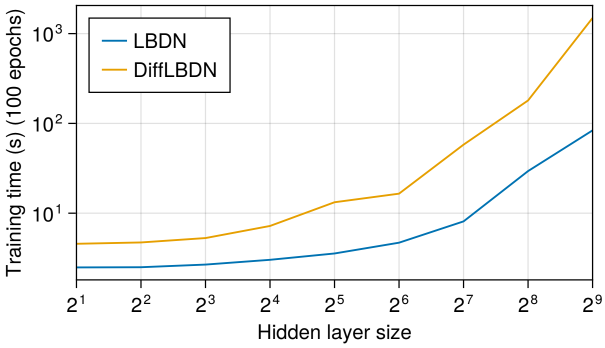

Fig. 8. Training time as a function of hidden-layer size for a single-hidden-layer LBDN constructed with both the `LBDN` and `DiffLBDN` wrappers. Using the `LBDN` wrapper for RL is significantly more efficient than re-constructing the explicit model at every evaluation of the `DiffLBDN` model.

3.3 Observer design

In Section 3.2, we designed a controller for a simple nonlinear system assuming that the controller had *full state knowledge*: that is, it had access to both the position and velocity of the box. In many practical situations, we may only be able to measure some of the system states. For example, our box may have a camera to estimate its position but not its velocity. In these cases, we need a *state observer* to estimate the full state of the system for feedback control.

In this example, we will show how a contracting REN can be used to learn stable observers for dynamical systems. A common approach to designing state estimators for nonlinear systems is the *Extended Kalman Filter* (EKF). In our case, we will consider observer design as a supervised learning problem. For a detailed explanation of the theory behind learning state observers, and for a similar example designing an observer for a *Partial Differential Equations* (PDE), please refer to Section VIII of [16].

3.3.1 Background theory. We briefly summarise some background theory from [16] relevant to this example. Suppose we have a discrete-time, nonlinear dynamical system of the form

$$x_{t+1} = f_d(x_t, u_t) \tag{14}$$
$$y_t = g_d(x_t, u_t) \tag{15}$$

with state vector x_t, controlled inputs u_t, and measured outputs y_t. Our aim is to estimate the sequence $\{x_0, x_1, \dots, x_T\}$ over some time period $[0, T]$ given only the measurements y_t and inputs u_t at each time step. We will use a very general form for an observer

$$\hat{x}_{t+1} = f_o(\hat{x}_t, u_t, y_t) \tag{16}$$

where $\hat{x}_t$ is the state estimate. A more common (but more restrictive) structure is the well-known Luenberger observer [12].

To estimate the true state, our observer error $(x_t - \hat{x}_t)$ must converge to zero as time progresses, or $\hat{x}_t \to x_t$ as $t \to \infty$. As outlined in [16], our observer only has to satisfy the following two conditions to guarantee this.

(1) The observer must be a contracting system (Sec. *2.2.1*).

(2) The observer must satisfy a "correctness" condition which says that, given perfect knowledge of the state, measurements, and inputs, the observer can exactly predict the next state. Mathematically, we write this as

$$f_o(x_t, u_t, y_t) = f_d(x_t, u_t) \tag{17}$$

where $y_t = g_d(x_t, u_t)$. Note the use of x_t not $\hat{x}_t$. It turns out that if the correctness condition is only approximately satisfied such that $|f_o(x_t, u_t, y_t) - f_d(x_t, u_t)| < \rho$ for some small number $\rho \in \mathbb{R}$, then the observer error will still be bounded. See Appendix E of [16] for details.

The first condition, contraction, is already guaranteed for all REN models in `RobustNeuralNetworks.jl`. Therefore, to learn a stable observer with RENs, our only requirement is to minimise the one-step-ahead prediction error to approximate the correctness condition. If we have a batch of data $z = \{x_i, u_i, y_i,\ i = 1, 2, \dots, N\}$, this corresponds to minimising the loss function

$$\mathcal{L}(z, \theta) = \sum_{i=1}^{N} |f_o(x_i, u_i, y_i) - f_d(x_i, u_i)|^2, \tag{18}$$

where θ contains the learnable parameters of the REN.

3.3.2 Generate training data. Consider the same nonlinear box system from Section 3.2, with the a change in setup so that we can only measure the box position. We introduce a measurement function `gd()` such that $y_t = x_t$.

```julia
m = 1                        # Mass (kg)
k = 5                        # Spring constant (N/m)
μ = 0.5                      # Viscous damping (kg/m)
nx = 2                       # Number of states

f(x::Matrix,u::Matrix) = [x[2:2,:]; (u[1:1,:] -
    k*x[1:1,:] - μ*x[2:2,:]*abs.(x[2:2,:]))/m]
fd(x,u) = x + dt*f(x,u)
gd(x::Matrix) = x[1:1,:]
```

For this example, we assume that the box always starts at rest in a random initial position between ±0.5m, after which it is released and allowed to oscillate freely with no added forces (so $u = 0$). Learning an observer typically requires a large amount of training data to fully capture the behaviour of the system, hence we consider 200 batches each simulating 10 s of motion.

```julia
Tmax = 10                   # Simulation horizon
dt = 0.01                   # Time step (s)
ts = 1:Int(Tmax/dt)         # Time array indices

# Generate batches of training data
batches = 200
u = fill(zeros(1, batches), length(ts)-1)
X = fill(zeros(1, batches), length(ts))
X[1] = 0.5*(2*rand(nx, batches) .- 1)

for t in ts[1:end-1]
    X[t+1] = fd(X[t],u[t])
end
```

We have stored the states of the system across each batch in `X`. To compute the one-step-ahead loss $\mathcal{L}$, we will need to separate this data into the states at the "current" time step `Xt` and at the "next" time step `Xn`, then compute the measurement outputs. We then store the data for training, shuffling it so there is no bias in the training towards earlier time steps.

```julia
using Random

# Current/next state, measurements
Xt = X[1:end-1]
Xn = X[2:end]
y  = gd.(Xt)

# Store training data
obsv_data = [[ut; yt] for (ut,yt) in zip(u, y)]
indx = shuffle(1:length(obsv_data))
data = zip(Xn[indx], Xt[indx], obsv_data[indx])
```

3.3.3 Define a model. We can construct the parameterization for a contracting REN model using `ContractingRENParams`. The inputs to the model are $[u_t; y_t]$, and its outputs are the next state estimate $\hat{x}_{t+1}$. The flag `output_map=false` sets the output map of the REN to just return its own internal state – i.e., $C_2 = I$, $D_{21} = 0$, $D_{22} = 0$, $b_y = 0$ from Equation 4. This makes the internal state of the REN exactly the state estimate $\hat{x}_t$.

```julia
using RobustNeuralNetworks

T  = Float32
nv = 200
nu = size(obsv_data[1], 1)
ny = nx
model_ps = ContractingRENParams{T}(
    nu, nx, nv, ny; output_map=false)
model = DiffREN(model_ps)
```

3.3.4 Define a loss function. As outlined in Section *3.3.1*, our loss function should be the one-step-ahead prediction error of the REN observer. We write this as follows, noting that all subtypes of `AbstractREN` return both their updated internal state and their output (in that order).

```
using Statistics

function loss(model, xn, xt, inputs)
    xpred = model(xt, inputs)[1]
    return mean(sum((xn - xpred).^2, dims=1))
end
```

3.3.5 Train the model. The function below trains the observer with the `Adam` optimizer over 100 epochs and decreases the maximum learning rate from 10^{-3} to 10^{-4} if the mean loss stops decreasing between epochs. The core of this function is a simple `Flux.jl` training loop, expanded out for clarity.

```
using Flux

function train_observer!(
    model, data;
    epochs=50, lr=1e-3, min_lr=1e-6
)
    opt_state = Flux.setup(Adam(lr), model)
    mean_loss = [1e5]
    for epoch in 1:epochs

        # Gradient descent update
        batch_loss = []
        for (xn, xt, inputs) in data
            tloss, dJ = Flux.withgradient(
                loss, model, xn, xt, inputs)
            Flux.update!(opt_state, model, dJ[1])
            push!(batch_loss, tloss)
        end

        # Reduce lr if loss is stuck or growing
        push!(mean_loss, mean(batch_loss))
        if (mean_loss[end] >= mean_loss[end-1]) &&
           (lr > min_lr)
            lr *= 0.1
            Flux.adjust!(opt_state, lr)
        end
    end
    return mean_loss
end
tloss = train_observer!(model, data)
```

3.3.6 Evaluate the trained model. We have trained the REN observer to minimise the one-step-ahead prediction error, but we are yet to test whether the the observer error actually does converge to zero. We set up the following 50 batches of test data as a demonstration.

```
batches   = 50
ts_test   = 1:Int(20/dt)
u_test    = fill(zeros(1, batches),length(ts_test))
x_test    = fill(zeros(nx,batches),length(ts_test))
x_test[1] = 0.2*(2*rand(nx, batches) .-1)

for t in ts_test[1:end-1]
    x_test[t+1] = fd(x_test[t], u_test[t])
end
y_test = gd.(x_test)
obsv_in = [[u;y] for (u,y) in zip(u_test, y_test)]
```

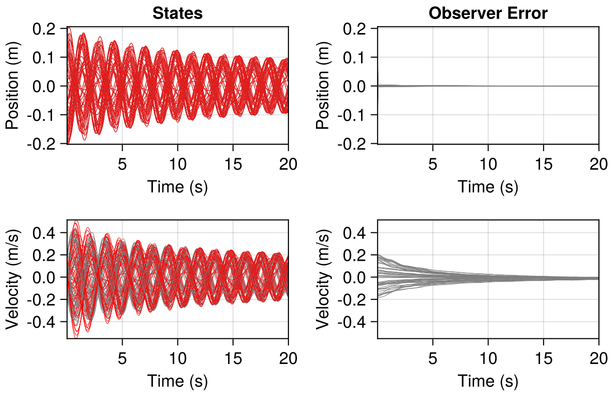

Fig. 9. Simulation results showing the observer predictions and observer error with the box starting at 50 different initial conditions. The left panels compare the true (grey) and estimated (red) states, while the right panels show the observer error $x - \hat{x}$ over time. The observer error converges for all 50 test cases.

Next, we need a function to simulate the REN observer using its own state $\hat{x}_t$ rather than the true system state x_t, which was used for training. We use the very neat tool `Flux.Recur` for this. We assume that the observer has no knowledge of the initial state and simply guesses $\hat{x}_0 = 0$ for all 50 batches.

```
function simulate(model::AbstractREN, x0, u)
    recurrent = Flux.Recur(model, x0)
    output = recurrent.(u)
    return output
end
x0hat = zeros(model.nx, batches)
xhat = simulate(model, x0hat, obsv_in)
```

The results are plotted in Figure 9. In the left-hand panels, the observer predictions (red) almost exactly match the true states (grey) after approximately 4 s. This is confirmed by the right-hand panels, which show the observer error $x_t - \hat{x}_t$ smoothly converging to zero as the observer estimates the correct states for all simulations.

It is worth noting that at no point did we directly train the REN to minimise the observer error. This is a natural result of using a model that is guaranteed to be contracting, and training it to minimise the one-step-ahead prediction error. There is still some residual observer error in the velocity in Figure 9, since our observer was only trained to approximately satisfy the correctness condition. However, this could easily be reduced or eliminated using a larger observer model, more training data, and more training epochs.

4. Summary and conclusions

This paper has presented `RobustNeuralNetworks.jl`, a Julia package for robust machine learning based on the recently-proposed Recurrent Equilibrium Network (REN) and Lipschitz-Bounded Deep Network (LBDN) model classes. The models are unique in that they naturally satisfy a set of *built-in* robustness certificates, such as contraction and Lipschitz bounds. We have presented an overview of the model architectures, including background theory on robustness metrics in nonlinear systems, and have outlined the package structure and its usage alongside Julia's main machine-learning library, `Flux.jl`. We have demonstrated via examples in image classification, reinforcement learning, and observer design that the package is easy to use in many common machine learning and data-driven control problems, while also offering the advantage of robustness guarantees.

We intend `RobustNeuralNetworks.jl` to be widely-applicable in the scientific and machine learning communities for learning-based problems in which robustness certificates are crucial, and have already used the package in our own research in robust reinforcement learning [1]. Some areas in which this package will be most applicable include: data-driven control and state estimation, image classification and segmentation, and privacy and security. We intend to expand the package with more robust neural network architectures in the future. Examples include LBDNs with one-dimensional convolution [14] and circular convolutions [23], continuous-time REN models [13], and RENs respecting other non-Euclidean contraction metrics [4]. We encourage any and all contributions to `RobustNeuralNetworks.jl` to further its use in robust machine learning problems.

Acknowledgements. This work was supported in part by the Australian Research Council (DP190102963).

5. References

[1] Nicholas H. Barbara, Ruigang Wang, and Ian R. Manchester. Learning over contracting and lipschitz closed-loops for partially-observed nonlinear systems. *2023 62nd IEEE Conference on Decision and Control (CDC)*, pages 1028–1033, 12 2023. doi:10.1109/CDC49753.2023.10383269.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59:65–98, 2017. doi:10.1137/141000671.

[3] Francesco Bullo. *Contraction Theory for Dynamical Systems*. Kindle Direct Publishing, 1.0 edition, 2022.

[4] Alexander Davydov, Saber Jafarpour, and Francesco Bullo. Non-euclidean contraction theory for robust nonlinear stability. *IEEE Transactions on Automatic Control*, 67:6667–6681, 12 2022. doi:10.1109/TAC.2022.3183966.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-December:770–778, 12 2016. doi:10.1109/CVPR.2016.90.

[6] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. International Conference on Learning Representations, ICLR, 2017.

[7] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018.

[8] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018. doi:10.21105/joss.00602.

[9] Neelay Junnarkar, He Yin, Fangda Gu, Murat Arcak, and Peter Seiler. Synthesis of stabilizing recurrent equilibrium network controllers. *Proceedings of the IEEE Conference on Decision and Control*, 2022-December:7449–7454, 2022. doi:10.1109/CDC51059.2022.9992684.

[10] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. International Conference on Learning Representations, ICLR, 12 2015.

[11] Yann LeCun, Corinna Cortes, and C J Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[12] David G. Luenberger. An introduction to observers. *IEEE Transactions on Automatic Control*, 16:596–602, 1971. doi:10.1109/TAC.1971.1099826.

[13] Daniele Martinelli, Clara Lucía Galimberti, Ian R. Manchester, Luca Furieri, and Giancarlo Ferrari-Trecate. Unconstrained parametrization of dissipative and contracting neural ordinary differential equations. *Proceedings of the IEEE Conference on Decision and Control*, pages 3043–3048, 2023. doi:10.1109/CDC49753.2023.10383704.

[14] Patricia Pauli, Dennis Gramlich, and Frank Allgöwer. Lipschitz constant estimation for 1d convolutional neural networks. *Proceedings of Machine Learning Research*, 211:1–14, 11 2022.

[15] Patricia Pauli, Anne Koch, Julian Berberich, Paul Kohler, and Frank Allgower. Training robust neural networks using lipschitz bounds. *IEEE Control Systems Letters*, 6:121–126, 2022. doi:10.1109/LCSYS.2021.3050444.

[16] Max Revay, Ruigang Wang, and Ian R. Manchester. Recurrent equilibrium networks: Flexible dynamic models with guaranteed stability and robustness. *IEEE Transactions on Automatic Control*, pages 1–16, 2023. doi:10.1109/TAC.2023.3294101.

[17] Alessio Russo and Alexandre Proutiere. Towards optimal attacks on reinforcement learning policies. *Proceedings of the American Control Conference*, 2021-May:4561–4567, 5 2021. doi:10.23919/ACC50511.2021.9483025.

[18] Jonah Siekmann, Yesh Godse, Alan Fern, and Jonathan Hurst. Sim-to-real learning of all common bipedal gaits via periodic reward composition. *Proceedings - IEEE International Conference on Robotics and Automation*, 2021-May:9943–9949, 2021. doi:10.1109/ICRA48506.2021.9561814.

[19] Xujie Song, Jingliang Duan, Wenxuan Wang, Shengbo Eben Li, Chen Chen, Bo Cheng, Bo Zhang, Junqing Wei, and Xiaoming Simon Wang. LipsNet: A smooth and robust neural network with adaptive Lipschitz constant for high accuracy optimal control. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 32253–32272. PMLR, 23–29 Jul 2023.

[20] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[21] Jun Tian and other contributors. Reinforcementlearning.jl: A reinforcement learning package for the julia programming language. 2020. [Online]. Available: `https://github.com/JuliaReinforcementLearning/ReinforcementLearning.jl`.

[22] Ruigang Wang, Nicholas H. Barbara, Max Revay, and Ian R. Manchester. Learning over all stabilizing nonlinear controllers for a partially-observed linear sys-

tem. *IEEE Control Systems Letters*, pages 1–1, 2022. doi:10.1109/LCSYS.2022.3184847.

[23] Ruigang Wang and Ian Manchester. Direct parameterization of lipschitz-bounded deep networks. volume 202, pages 36093–36110. PMLR, 7 2023.

[24] Ruigang Wang and Ian R. Manchester. Youla-ren: Learning nonlinear feedback policies with robust stability guarantees. *Proceedings of the American Control Conference*, 2022-June:2116–2123, 2022. doi:10.23919/ACC53348.2022.9867842.

[25] Dante C. Youla, Joseph J. Bongiorno, and Hamid A. Jabr. Modern wiener-hopf design of optimal controllers — part ii: The multivariable case. *IEEE Transactions on Automatic Control*, 21:319–338, 1976. doi:10.1109/TAC.1976.1101223.