A Practical Introduction to Echo State Networks using Python

Ryan Hill*

January 2021

1 Formalism

The characteristic task of supervised learning with RNNs, and with all ANNs for that matter, is to learn a functional relation between a given input $\mathbf{u}(n) \in \mathbb{R}^{N_u}$ and a desired output $\bar{\mathbf{y}}(n) \in \mathbb{R}^{N_y}$, where n = 1,...,T, and T is the number of data points in the training dataset $(u(n), \bar{\mathbf{y}}(n))$. For a non-temporal task (i.e. data points are independent of each other) the goal is to learn a function $\mathbf{y}(n) = y(\mathbf{u}(n))$ such that the loss function $E(\mathbf{y}, \bar{\mathbf{y}})$ is minimized. In the context of RNNs, a common loss function is the normalized root-mean-square error (NRMSE):

$$E(\mathbf{y}, \bar{\mathbf{y}}) = \sqrt{\frac{\langle \|\mathbf{y}(n) - \bar{\mathbf{y}}(n)\|^2 \rangle}{\langle \|\mathbf{y}(n) - \langle \bar{\mathbf{y}}(n) \rangle\|^2 \rangle}} \tag{1}$$

A temporal task is where $\mathbf{u}$ and $\bar{\mathbf{y}}$ are signals in a discrete time domain (n = 1,...,T), and the goal is to learn a function $\mathbf{y}(n) = y(..., \mathbf{u}(n-1), \mathbf{u}(n))$ such that $E(\mathbf{y}, \bar{\mathbf{y}})$ is minimized. Thus, the difference between the temporal and non-temporal task is that the function $\mathbf{y}(n)$ we are trying to learn has memory in the first case and is memory-less in the second.

Many tasks cannot be accurately solved by a simple linear relation between $\mathbf{u}$ and $\bar{\mathbf{y}}$. In this case, we resort to a nonlinear model, which can be achieved by taking a nonlinear expansion of the input $\mathbf{u}$. In RC, the function of the reservoir is to act both as this nonlinear expansion and as a memory input. To perform a nonlinear high-dimensional expansion $\mathbf{x}(n) \in \mathbb{R}^{N_x}$ of the input signal $\mathbf{u}(n) \in \mathbb{R}^{N_u}$, we require $N_x >> N_u$. Input data which is not linearly separable in the original space $\mathbb{R}^{N_u}$ often becomes separable in the expanded space $\mathbb{R}^{N_x}$. We are therefore searching for solutions of the form

$$\mathbf{y}(n) = \mathbf{W}_{out}\mathbf{x}(n) = \mathbf{W}_{out}x(\mathbf{u}(n)) \tag{2}$$

*rjh324@cornell.edu

where $\mathbf{W}_{out} \in \mathbb{R}^{N_y \times N_x}$ are the trained output weights and $\mathbf{y}(n)$ is the the learned function. Expansion $\mathbf{x}(n)$ is often referred to as the "state vector" of the system at time step n. The network is usually initialized in state

$$\mathbf{x}(0) = \mathbf{0} \tag{3}$$

The reservoir also serves as memory, providing temporal context. This is a crucial reason for using RNNs in the first place. In a temporal task the function to be learned depends also on the history of the input. Thus, the expansion function has memory: $\mathbf{x}(n) = x(..., \mathbf{u}(n-1), \mathbf{u}(n))$. This function has an unbounded number of parameters, so we can express it recursively:

$$\mathbf{x}(n) = x(\mathbf{x}(n-1), \mathbf{u}(n)) \tag{4}$$

With the recursive (temporal) definition of the state vector, the output $\mathbf{y}(n)$ is (typically) still calculated in the same way as for non-temporal methods (2). For non-temporal tasks, this recursive definition can act as a type of a spatial embedding of temporal information. This enables learning of high-dimensional dynamical tendencies (or "attractors" [1]) of the system from low-dimensional observations. This is shown possible by Takens's theorem [2].

Combining the state vector non-linear expansion and memory components leads to the following general RNN state update equation,

$$\mathbf{x}(n) = f(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{5}$$

where f is the neuron activation function, usually symmetric tanh, applied element-wise, $\mathbf{W}_{in} \in \mathbb{R}^{N_x \times N_u}$ is the input weight matrix, and $\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$ is the internal (hidden) weight matrix of network connections.

The ESN is a recurrent neural network with a sparsely connected hidden layer (the reservoir), driven by a (one- or multi-dimensional) time signal, and applied to supervised temporal ML tasks. The connectivity and weights of hidden neurons are fixed and randomly assigned. The weights connecting the hidden neurons to the output neurons are the only trainable parameters in the network. Although ESNs are dynamic, with non-linear behavior, their "single-layer training" attribute makes their loss function quadratic in "parameter-space," so it can be differentiated to a linear system. Quadratic loss functions are desirable because they are easily manipulated (property of variances), symmetric, and allow easy application of linear regression.

ESNs are an attractive RC implementation method because they are conceptually simple and computationally inexpensive. However, creating an effective ESN is not, in and of itself, straightforward. The following sections will overview ESN implementation, training, and application. We will approach ESN implementation in the practical context of a simple Python3 program using the PyTorch open-source ML library. To do so, we will work through each of the "To Dos" in example code outline `class Reservoir()` and `class ESN(.)`.

In the code outlines (Figure 1, Figure 2), a question mark, "`?`", indicates a numerical value selected by the user. "TODO", abbreviated TD# from here on, indicates a section of code yet to be implemented, where # in $[1, 6]$ is the order of implementation followed in this paper.

```
import torch
import torch.nn as nn
from reservoir import Reservoir

class ESN(nn.Module):
  def __init__(self, input_size, output_size, hidden_size=?):
    super().__init__()
    self.reservoir = Reservoir(input_size, hidden_size)
    self.linear = nn.Linear(hidden_size,
      output_size, bias=False)
    self.softmax = nn.Softmax(dim=-1)

  def forward(self, u):
    # TODO 3
    x = self.reservoir(u)
    # TODO 6
    return y
```

Figure 1: esn.py

```
import torch
import random

class Reservoir:
  def __init__(self, input_size, res_size, sparsity=?,
  spect_rad=?, input_scale=?, leak_rate=?):
    self.input_size = input_size
    self.res_size = res_size
    self.sparsity = sparsity
    self.spect_rad = spect_rad
    self.input_scale = input_scale
    self.leak_rate  = leak_rate
    self.w_in = self.gen_w_in(self.res_size,
      self.input_size, self.input_scale)
    self.w = self.gen_w(self.res_size, self.spect_rad)
    self.state_x = self.init_state(self.res_size)

  def init_state(self, res_size):
    # TODO 4
    return init_state

  def gen_w_in(self, res_size, input_size, input_scale):
    # TODO 2
    return w_in

  def gen_w(self, res_size, spect_rad):
      # TODO 1
      return w

  def _forward(self, u):
    assert(u.size() == self.input_size)
    with torch.no_grad():
      # TODO 5
      self.state_x = updated_x
      return updated_x

  def __call__(self, u):
    return self._forward(u)
```

Figure 2: reservoir.py

2 Reservoir Generation

The reservoir is defined by tuple $(\mathbf{W}_{in}, \mathbf{W}, \alpha)$. $\mathbf{W}_{in}$ and $\mathbf{W}$ are generated randomly according to a number of hyperparameters (in analogy to other ANN approaches, "hyperparameters" refer to parameters governing the distribution of connection weights, as opposed to the connection weights themselves). α is the leaking rate. Reservoir hyperparameters must be subtly chosen, as each significantly impacts the behavior of the network, and therefore its overall performance. Characteristic reservoir hyperparameters include the reservoir size, sparsity, and distribution of nonzero elements, the spectral radius of $\mathbf{W}$, the scaling of $\mathbf{W}_{in}$, and, of course, the leaking rate α. These values are initialized in the `Reservoir` class constructor. In this implementation, the input size and desired reservoir size are set inside of the `ESN` class upon instantiation of a `Reservoir` object.

`gen_w(.)`

To begin, we wish to define a function `gen_w()` (TD1), which generates the random, internal weight matrix $\mathbf{W}$ of size $N_x \times N_x$ (5). The bigger the space of reservoir signals $\mathbf{x}(n)$, the easier it is to find a linear combination of the signals to approximate $\bar{\mathbf{y}}(n)$. Therefore, the bigger the reservoir, the better the obtainable performance. However, reservoir size *is* bounded by its memory capacity (number of values it must remember from the input to accomplish the task, lower bound) and by the over-fitting threshold (upper bound):

$$N_y \leq N_x \leq T - N_u \tag{6}$$

where T is the number of training data points.

Weight matrix $\mathbf{W}$ is typically sparse. Compared to dense representations, this enables faster state vector updates while also giving slightly better performance [3]. Nonzero elements can take any distribution, though common choices include uniform, discrete bi-valued, and normal [4]. There does not exist built-in Python functionality to construct, simultaneously, a matrix according to a given density (or sparsity) *and* a given nonzero element distribution. Therefore, these tasks must be performed in sequence. For example, using `torch.randn`, we can easily generate a random matrix of size N_x with values (weights) normally distributed around zero. The initial width of the distribution does not matter, as it is eventually reset according to the reservoir's spectral radius. We can then iterate over this generated matrix, drawing a random number $p \in [0.0, 1.0)$ at each entry $\mathbf{W}_{i,j}$ using Numpy library function `random.random()`. For desired sparsity $s \in (0.5, 1.0)$, if $p_{i,j} \leq s$, we set $\mathbf{W}_{i,j} = 0$. Else, $\mathbf{W}_{i,j}$ is unchanged. This is merely one of many possible approaches.

Spectral radius is one of the ESN's most central hyperparameters; it specifies the largest absolute eigenvalue of weight matrix $\mathbf{W}$. After a random sparse $\mathbf{W}$ (now named $\mathbf{W}_0$) is generated, we can compute its spectral radius

$$\rho_0(\mathbf{W}_0) = \lambda_{max}(\mathbf{W}_0) \tag{7}$$

using PyTorch's `torch.eig` and `torch.max`, or equivalent functions. Next, we rescale $\mathbf{W}_0$ according to user-specified spectral radius, ρ (28):

$$\mathbf{W} = \frac{\rho}{|\rho_0|}\mathbf{W}_0 \tag{8}$$

Finally, `return` the rescaled weight matrix $\mathbf{W}$, now with largest absolute eigenvalue ρ, and `gen_w()` is complete.

For the ESN model to work, the reservoir must satisfy the echo-state property (ESP): the effect of previous state $\mathbf{x}(n)$ and previous input $\mathbf{u}(n)$ on a future state $\mathbf{x}(n+k)$ should vanish gradually as time passes ($k \to \infty$) [5]. In other words, the reservoir must asymptotically "wash out" any information from initial conditions. For reservoirs with tanh activation and input $\mathbf{u}(n) = \mathbf{0}$, the ESP is *violated* if $\mathbf{W}$ is scaled such that

$$\rho(\mathbf{W}) > 1 \tag{9}$$

Contrary to many simplistic descriptions, $\rho < 1$ does not guarantee the ESP. The ESP can be obtained for $\mathbf{u}(n) \neq \mathbf{0}$ even if $\rho > 1$, and can be violated even if $\rho < 1$ [5] (although the latter is unlikely). Through a process of trial and error, ρ should be selected to maximize performance. Generally, ρ should be close to 1 for tasks that require long term memory, and smaller for tasks where too much memory may be harmful.

We have now constructed internal weight matrix $\mathbf{W}$ according to user-specified hyperparameters reservoir size, sparsity, distribution of nonzero elements, and spectral radius.

`gen_w_in(.)`

Following a similar procedure, we will now define a function `gen_w_in()` (TD2), which generates the random, input weight matrix $\mathbf{W}_{in}$ of size $N_x \times N_u$ (5). N_x is the reservoir size, and was specified while generating $\mathbf{W}$. N_u is the dimensionality of the input data, so is also predetermined. $\mathbf{W}_{in}$ is usually generated according to the same distribution as $\mathbf{W}$. Therefore, a normal distribution of weights around 0 may, once again, be carried out using the `torch.randn` function. This time, however, the width of the distribution *does* does matter: for normal distributed input weights we can take the standard deviation as a scaling measure [4]. This scaling measure, known as the input scaling, a, is a vital ESN optimization parameter. It dictates how "nonlinear" reservoir responses are.

For small input scaling values, the units of $\mathbf{W}_{in}$ will collapse towards 0, where their tanh activation is essentially linear (Figure 2). So for linear tasks, a should be small. For large input scaling values, the units of $\mathbf{W}_{in}$ will quickly saturate towards 1 and -1, exhibiting nonlinear, binary "switching" behavior. So for nonlinear tasks, reservoir response *might* be optimized for larger a. The "amount" of nonlinearity a task requires is not always easy to assess. This hyperparameter is another best tuned through trail and error.

Referring back to the general state update equation (5), it is clear, for sigmoid activation, that the scaling a of $\mathbf{W}_{in}$, and the scaling $\rho(\mathbf{W})$ of $\mathbf{W}$, together

dictate the amount of nonlinearity and memory of previous states present in the current state representation.

To implement `gen_w_in()`, sequentially construct $\mathbf{W}_{in}$ by first creating an $N_x \times N_u$ dimensional matrix with the same element distribution as $\mathbf{W}$. But this time, form the distribution according to the input scaling, a: For uniform distributions, sample values from the interval $[-a, a]$. For normal distributions, set $\sigma = a$. Then, iterate through the resulting matrix, drawing random number $p \in [0.0, 1.0)$ at each entry, and setting $\mathbf{W}_{in_{i,j}} = 0$ where $p_{i,j} \leq s$. $\mathbf{W}_{in}$ is typically dense, so $s \in [0.0, 0.5)$. `return` $\mathbf{W}_{in}$ and `gen_w_in()` is complete.

We have now constructed input weight matrix $\mathbf{W}_{in}$ according to the input data dimension, reservoir size, nonzero element distribution of $\mathbf{W}$, and user-specified input scaling parameter. In the next section, we will further specify our reservoir model by covering neuron activation now in the context of RNNs.

3 Activation States

This section will overview the leaky integrator neuron model, and then describe the computation and collection of reservoir activation states $\mathbf{x}(n)$ for a training input $\mathbf{u}(n)$.

`forward(.)` Pt. 1

In ANN training, the "forward pass" refers to the computation performed during each training iteration. In a single forward pass, the network is provided one training data sample whose information is propagated through the hidden units of the network to eventually produce an output $\mathbf{y}$, at which point a linear readout is performed. In the first component of the `class ESN` "forward" method (TD3) we manipulate the input, $\mathbf{u}$, to match the user-specified reservoir input dimension. You can read-in data any way you like, as along as all of the input information is inside the reservoir simultaneously. Take, for example, an input of dimension $1 \times N \times N$. A natural way to perform this read-in would be to generate a reservoir $\mathbf{W}$ of $N \times N$ neurons, and to flatten and transpose each input into a $N^2 \times 1$ column vector. This could be achieved through the `torch.flatten()` and `torch.t()` methods. However, you may alternatively rely on the fact that the reservoir has a memory, and read-in each single column of the input as a $N \times 1$ vector at each time point, ultimately taking N time points to get the entire input into the reservoir.

`init_state`(.)

Before we can use activation to perform a state update, we need an initial state (TD4). The reservoir state vector must be of dimension $N_x \times 1$ (5). If we have chosen a spectral radius such that our reservoir satisfies the ESP ((9) *not* true), then the final state of the network does not depend on its initial conditions. Therefore, the initial state can be any vector of the above dimension. A com-

mon choice is the zero vector (3), which can be created using the `torch.zeros` function.

`forward(.)` Pt. 2

As previously stated, standard ESNs use sigmoid neurons, i.e., reservoir states are computed using (5), where the nonlinear function f is a sigmoid, usually the tanh function:

$$\mathbf{x}(n) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{10}$$

Leaky integrator neuron models represent another frequent option for ESNs. These type of neurons perform a "leaky" integration of their activation from previous time steps. Generally, a leaky integrator is analogous to a first-order filter with feedback.

For arbitrary input $\mathbf{u}(n)$ and output $\mathbf{y}(n)$, a leaky integrator can be described by nonhomogeneous first-order linear ODE

$$\dot{\mathbf{y}}(n) + \alpha\mathbf{y}(n) = \mathbf{u}(n) \tag{11}$$

where α is the leaking rate. Using (9), let us redefine

$$\mathbf{u}(n) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{12a}$$
$$\alpha\mathbf{y}(n) = \mathbf{x}(n) \tag{12b}$$

Plugging our substitutions (11) into (10) and rearranging,

$$\dot{\mathbf{x}}(n) + \mathbf{x}(n) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{13}$$

$$\dot{\mathbf{x}} = -\mathbf{x} + \tanh(\mathbf{W}_{in}\mathbf{u} + \mathbf{W}\mathbf{x}) \tag{14}$$

where (13) describes the continuous time reservoir update dynamics. Performing an Euler's discretization in time on (13),

$$\dot{\mathbf{x}} \approx \frac{\Delta\mathbf{x}}{\Delta t} = \frac{\mathbf{x}(n+1) - \mathbf{x}(n)}{\Delta t} \tag{15}$$

For brevity, we will redefine the right hand side of (12):

$$\tilde{\mathbf{x}}(n+1) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{16}$$

Setting (20) equal to (19), and using simple algebra:

$$\frac{\mathbf{x}(n+1) - \mathbf{x}(n)}{\Delta t} = -\mathbf{x}(n) + \tilde{\mathbf{x}}(n+1) \tag{17}$$

$$\mathbf{x}(n+1) - \mathbf{x}(n) = \Delta t(-\mathbf{x}(n) + \tilde{\mathbf{x}}(n+1)) \tag{18}$$

$$\mathbf{x}(n+1) = (1 - \Delta t)\mathbf{x}(n) + \Delta t\tilde{\mathbf{x}}(n+1)) \tag{19}$$

The leaking rate $\alpha \in (0, 1]$ of the reservoir nodes denotes the speed of the reservoir update dynamics *discretized in time*. Therefore, we can substitute

α for the sampling interval Δt. And finally, by performing time-translation $n \to n-1$, we arrive at the leaky-integrated discrete-time continuous-value unit ESN state update equations:

$$\tilde{\mathbf{x}}(n) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{20a}$$

$$\mathbf{x}(n) = (1-\alpha)\mathbf{x}(n-1) + \alpha\tilde{\mathbf{x}}(n) \tag{20b}$$

where $\mathbf{x}(n)$ is the current state vector and $\tilde{\mathbf{x}}(n)$ is its update, each at time step n. To apply a bias b (as in $y = mx + b$), simply use

$$\mathbf{u}(n) \to [b; \mathbf{u}(n)] \tag{21}$$

where $[\cdot;\cdot]$ stands for a vertical vector (or matrix) concatenation [4]. Notice, to use the model without leaky integration, simply set $\alpha = 1$ and thus $\tilde{\mathbf{x}}(n) \equiv \mathbf{x}(n)$.

We are now equipped to implement the second component of the `class ESN(.)` "forward" method, i.e. the "forward" method of `class Reservoir()`, (TD5): `_forward()` updates and returns the state representation $\mathbf{x}(n)$ for given input $\mathbf{u}(n)$. The single leading underscore is a convention used to indicate that this method is meant only for internal use. Here, it also used to differentiate between `forward()` of `class ESN(.)`, which is where the readout, and remainder of the "forward pass" will be performed (Section 4). Abstracting the code in this way, organizing methods acting in and on the reservoir into a separate `Reservoir` class, will prove exceptionally useful in debugging, parameter selection, and extending to multi-reservoir networks (DeepESNs).

Notice the global variable `state_x` inside of the `Reservoir` constructor method. This variable stores the state update of the previous training iteration (or pass), $\mathbf{x}(n-1)$ so it can be applied to perform a new update during the current pass. To implement TD5, use the built-in `torch.tanh` and `torch.mm` (matrix multiply) methods to evaluate (20). Store the result in a temporary variable, and evaluate (21) to produce `updated_x`. Finally, we update global variable `state_x` for the next pass, and return the updated state.

We have now created methods to generate an initial state $\mathbf{x}(0) = \mathbf{0}$, and to perform a leaky-integrated state update for given input $\mathbf{u}(n)$ using our previously generated $\mathbf{W}$ and $\mathbf{W}_{in}$, and globally stored $\mathbf{x}(n-1)$.

4 Compute a Probability Distribution

Restating (2) for convenience, our main objective is to find an output weight matrix $\mathbf{W}_{out}$ in $\mathbb{R}^{N_x}$ such that

$$\mathbf{W}_{out}\mathbf{x}(n) = \mathbf{y}(n) \approx \bar{\mathbf{y}}(n) \tag{22}$$

For tasks where feedback or temporal sequential processing is important, it often helps significantly to include the original input $\mathbf{u}(n)$ and a bias, b. In this case, the linear readout becomes

$$\mathbf{W}_{out}[b; \mathbf{u}(n); \mathbf{x}(n)] = \mathbf{y}(n) \tag{23}$$

where $\mathbf{y}(n) \in \mathbb{R}^{N_y}$, $\mathbf{W}_{out}(n) \in \mathbb{R}^{N_y \times (1+N_u+N_x)}$, and $[\cdot;\cdot;\cdot]$ again stands for a vertical vector (or matrix) concatenation [4].

Since the readouts from an ESN are typically linear and feedforward [4], learning the output weights (23) can be viewed as solving a system of linear equations

$$\mathbf{W}_{out}\mathbf{X} = \bar{\mathbf{Y}} \tag{24}$$

where $\mathbf{X} \in \mathbb{R}^{N \times T}$ are all $\mathbf{x}(n)$ produced by presenting the reservoir with $\mathbf{u}(n)$, and $\bar{\mathbf{Y}} \in \mathbb{R}^{N_y \times T}$ are all $\bar{\mathbf{y}}(n)$, both collected intro respective matrices over the training period $n = 1, ..., T$. Note: for linear readout using (23) and (24) respectively,

$$\mathbf{X} \in \mathbb{R}^{N \times T} \rightarrow \mathbf{X} \in \mathbb{R}^{N_x \times T} \tag{25a}$$

$$\mathbf{X} \in \mathbb{R}^{N \times T} \rightarrow [\mathbf{B};\ \mathbf{U};\ \mathbf{X}] \in \mathbb{R}^{(1+N_u+N_x) \times T} \tag{25b}$$

The goal is to minimize the (quadratic) error-rate $\mathbf{E}(\bar{\mathbf{Y}}, \mathbf{W}_{out}\mathbf{X})$ as in (1). This problem seems familiar from our previous discussion of supervised learning (Section (**??**)), so let's first approach it as we know how, using gradient descent (GD).

`forward(.)` Pt. 3

The third and final component of the `class ESN(.)` "forward" method (TD6) aims to use the state update(s) (20) computed in `_forward(.)` (TD5) to generate output predictions, $\mathbf{y}(n)$. In accordance with GD, we can then compare these predictions against true values $\bar{\mathbf{y}}(n)$ to iteratively update the weights of $\mathbf{W}_{out}$ in direction opposite to the gradient of the error function (1). Due to the auto-feedback nature of ESNs (i.e. their "echo" property), the reservoir state $\mathbf{x}(n)$ holds traces of past activations $\mathbf{x}(n-1)$, $\mathbf{x}(n-2)$, ..., etc. [1] It is therefore often sufficient to take $\mathbf{x}(n)$ alone as a representative encoding of the input history, and compute $\mathbf{y}(n)$ according to (22). We can do so using the PyTorch `nn.Linear` class, which applies a linear transformation to incoming data

$$\mathbf{y} = \mathbf{x}\mathbf{A}^\intercal + \mathbf{b} \tag{26}$$

Note: for some tasks (e.g. classification) performance is improved using time-averaged activations, $\sum \mathbf{x}$ [4]. But for now, we will proceed with just $\mathbf{x}(n)$ for clarity. In the `ESN` class definition (Figure 1) we instantiated `nn.Linear` such that (26) takes $\mathbf{x} \in \mathbb{R}^{N_x} \rightarrow \mathbf{y} \in \mathbb{R}^{N_y}$, as required. Setting `bias=False` ensures $\mathbf{b} = \mathbf{0}$ so that the only learnable parameters are the weights of $\mathbf{W}_{out}$ (i.e. elements of $\mathbf{A}^\intercal$). In TD6, transpose just-computed state update $\mathbf{x}(n)$ as necessary

[1] The extent of this dynamical short-term memory can be calculated explicitly, and is called the ESN's memory capacity, MC. It has been shown [6] that $MC \leq N$ for networks with linear output units and independent and identically distributed (iid) input, where N is the reservoir size.

and pass it through the linear layer. Finally, apply the `nn.Softmax` function, normalizing the linear layer resultant vector into a probability distribution, i.e. rescaling its elements such that they fall in the range $[0, 1]$ and sum to 1:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{27}$$

Many prefer Softmax over other normalization functions (e.g. argmax, standard normalization) because it is differentiable (useful for backprop, but not applicable in our case) and is exponential, emphasizing the extremes of the distribution. After returning the result of Softmax normalization `forward(.)` is fully implemented.

Our PyTorch ESN model (Figure 2) is now complete. We started by implementing methods to generate a reservoir $(\mathbf{W}_{in}, \mathbf{W}, \alpha)$ according to user-specified hyperparameters reservoir size, sparsity, distribution of nonzero elements, spectral radius, input scaling, and leaking rate. We then initialized the reservoir state vector, and finally, implemented a complete forward pass: we specified the read-in of input data, used that data to perform a tanh leaky-integrated state update, linearly transformed the resultant state according to weight matrix $\mathbf{W}_{out}$, and ultimately, produced a prediction in the form of a normalized probability distribution.

Remember, this is only one of *many* possible implementations. Each component of the network can (and should be) adapted to fit the demands of each unique learning task. However, the class structure, method specification, and general abstraction used in this paper should be intuitive and naturally extensible for anyone just starting out.

5 Iterative Optimization

In Section 4 we formulated a goal to solve the linear system of equations given by (24) or, equivalently, minimize the error-rate $\mathbf{E}(\bar{\mathbf{Y}}, \mathbf{W}_{out}\mathbf{X})$. First, we took the gradient descent approach. Now, for a given input $\mathbf{u}(n)$, our ESN generates a prediction $\mathbf{y}(n)$ in the form of an $N_y \times T$ dimensional probability distribution.

Next, we need to define an error (or loss) function. Two common, predefined choices in PyTorch are `nn.MSELoss()` and `nn.CrossEntropyLoss()`. Both loss functions have explicit probabilistic interpretations: MSE corresponds to estimating the mean of a distribution; cross-entropy corresponds to minimizing the negative log likelihood of a distribution (i.e. maximizing the log likelihood). MSE is thus better suited for regression (narrower, smoother decision boundary where the goal is to be close), while cross-entropy is preferred for classification (larger, more abrupt decision boundary where the penalty for being wrong increases exponentially as you get closer to predicting the wrong output.)

For a PyTorch optimizer, a good choice is Adam (`optim.Adam(.)`). First published in 2014, Adam is an algorithm for first-order gradient-based optimization of stochastic objective functions (randomness present), based on adaptive

estimates of lower-order moments (measures of the "shape" of a function) [7]. Adam can be viewed as a combination of RMSprop and Stochastic Gradient Descent (SGD) with momentum. Like RMSprop, it uses the squared gradients to scale the learning rate, and like SGD with momentum, it uses the moving average of the gradient instead of gradient itself.

Before training, we can quickly verify that the network is generating the correct number of learnable parameters. Assuming all the hyperparameters of `class Reservoir()` are set, specify a hidden size $\mathtt{R} \in \mathbb{Z}$ for `class ESN(.)`. Now we create an instance

```
esn = ESN(N, M)
```
(28)

where $\mathtt{N}, \mathtt{M} \in \mathbb{Z}$ are the input and output size, respectively. Finally, executing

```
params = list(esn.parameters())
print(len(params))
print(params[0].size())
```

should result in output

```
1
torch.Size([M, R])
```

As some back-of-the-envelope linear algebra, or a glance at (2) will prove, $\mathtt{M} \times \mathtt{R}$ is the correct dimension for trained weight matrix $\mathbf{W}_{out}$.

With our ESN model complete, error and optimizer functions selected and defined, and learned parameters correctly reflecting the dimension of $\mathbf{W}_{out}$, we are ready to perform gradient descent. This process is relatively straight forward and well documented on the PyTorch website, so will be omitted for the purposes of this paper.

6 Closed-Form Solutions

Using an alternative approach, we can evaluate (24) for $\mathbf{W}_{out}$ algebraically to find an exact, or closed-form solution. Typically $T \gg N$, so (24) presents an overdetermined system of linear equations:

$$\begin{bmatrix} & & \\ & \vdots & \\ & & \end{bmatrix} \begin{bmatrix} \mathbf{x}_1(n) \\ \vdots \\ \mathbf{x}_N(n) \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1(n) \\ \\ \vdots \\ \\ \mathbf{y}_{N_y}(n) \end{bmatrix} \tag{29}$$

Table 1: ESN & Training Parameters

Parameter	Value
Reservoir size (N_x)	1000
Spectral radius (ρ)	0.99
Input scaling (a)	0.6
Reservoir density (d_w)	0.1
Input density ($d_{w_{in}}$)	1.0
Leak rate (α)	1.0
Bias (b)	1
Training length (T)	8000

To be clear, the dimensionality of this equation follows $(N_y \times N) \cdot (N \times T) = (N_y \times T)$ where N is given by choice of linear readout using state vectors with or without vertical concatenation of input and bias (25) (or just one or the other; not given but easily deduced). The method for finding least square solutions of overdetermined systems of linear equations is also known as linear regression. Solutions following this method are outside the scope of this paper, but could be a subject of future work.

7 Application

The first application of our ESN was towards traditional MNIST classification. Each MNIST image is delivered as a 3-dimensional tensor (2 spacial dimensions, 1 RGB dimension). By transposing each image into a $28^2 \times 1$ column vector, we could deliver all of the information contained in an image to the reservoir at once (TD3). Figure 3 is a depiction of the performance of our network over 15 training iterations, or epochs.

The data shows us that for careful choice of parameters, we can apply an echo-state system (typically suited for temporal tasks) to a classification problem with very respectable results.

The reservoir did not respond well to leaky integration, because the MNIST images bear no temporal connection, so we set α to 1. The most impactful parameter on performance was the reservoir size. Over all of the training, there seemed no upper to limit to where a larger reservoir would not improve performance, besides what the computer could computationally handle. Though, the marginal benefit of continuing to increase reservoir size slowed to a near plateau for sizes roughly equivalent to that of the training data set. There may have been an over-fitting threshold (6), though the reservoir was still improving at larger sizes, just quadratically more slowly. A complete list of ESN and training parameters used during this readout are given in Table 1.

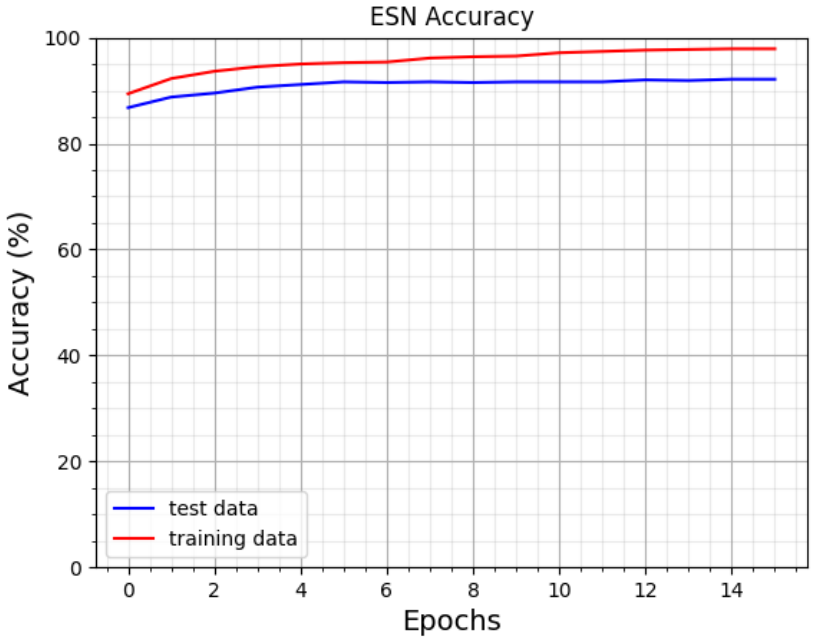

Figure 3: Accuracy of ESN of dimension $N_x = 1000$ on MNIST

References

[1] Miller, P. Dynamical systems, attractors, and neural circuits. *F1000Research*, 5:992, 2016.

[2] Floris Takens. Comparative analysis of recurrent and finite impulse response neural networks in time series prediction. *Lecture Notes in Mathematics Dynamical Systems and Turbulence, Warwick 1980*, pages 366–381, 1981.

[3] Erich Narang, Sharan an Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks, 2017.

[4] M. Lukoševičius. A Practical Guide to Applying Echo State Networks. *Lecture Notes in Computer Science Neural Networks: Tricks of the Trade*, pages 659–686, 2012.

[5] M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3:127–149, 2009.

[6] H. Jaeger. Short term memory in echo state networks. Technical report, German National Research Center for Information Technology, 2002.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.