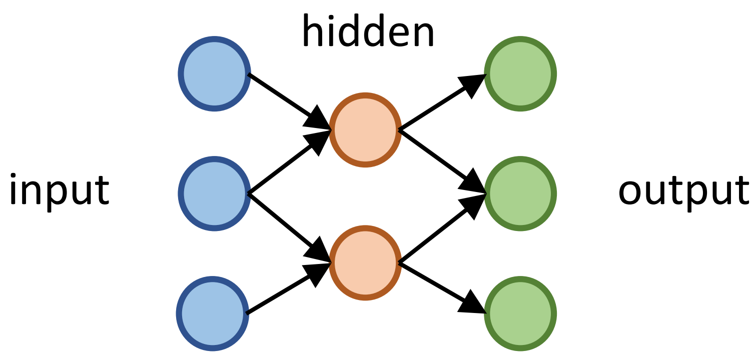
Introduction to Recurrent Neural Nets

Matt Smart
January 2020

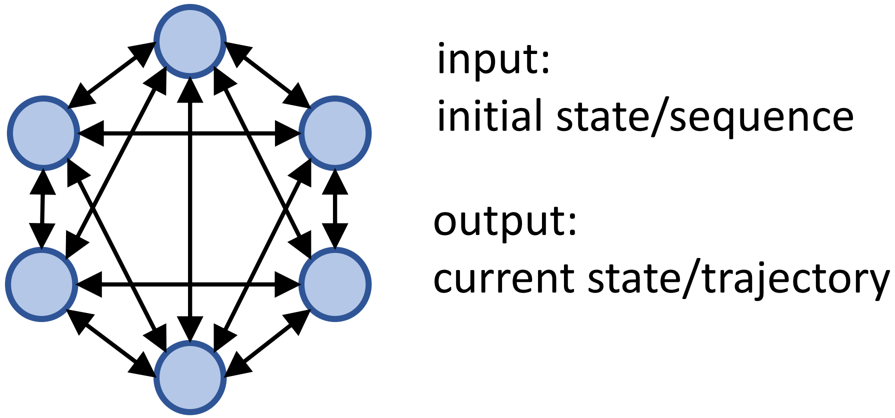
Feed-Forward Neural Nets



- Clear input to output direction
- Each layer transforms its input and passes it forward

- Objective functions built on output alone; don't typically track the intermediate layers

Recurrent Neural Nets



- Nodes pass information to and from each other as a connected graph
- No prescribed input to output direction

- Objective functions built around the RNN dynamics (trajectories, steady state distribution)

Feed-Forward Neural Nets

Universal Approximation Theorem

Comes in various forms. Roughly:

Any smooth function $f(\boldsymbol{x}): \mathbb{R}^n \to \mathbb{R}$ can be approximated to arbitrary accuracy by a FFNN with a single hidden layer, i.e.

$$F(\boldsymbol{x}) = \sum v_i \sigma\left(\boldsymbol{w}_i^T \boldsymbol{x} + b_i\right)$$

so that $\forall\ \boldsymbol{x} \in \mathbb{R}^n$

$$\left| F(x) - f(x) \right| < \epsilon$$

Recurrent Neural Nets

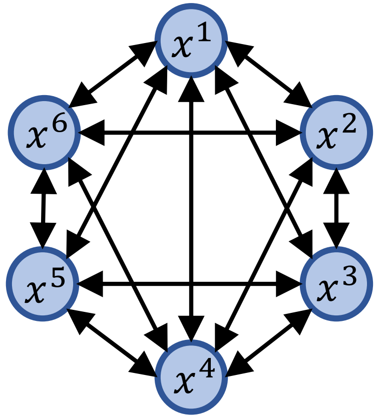
RNN generalization of UAT

Many variants. An early one: Theorem 2 of *Funahashi and Nakamura, 1993*

Any continuous-time, smooth dynamical system $\frac{dx}{dt} = \boldsymbol{F}(\boldsymbol{x})$ can be approximated (i.e., its trajectories) to arbitrary accuracy by a continuous-time RNN of the form

$$\boldsymbol{\tau}\frac{d\boldsymbol{x}}{dt} = -\boldsymbol{x} + \boldsymbol{W}\sigma(\boldsymbol{x}) + \boldsymbol{h}$$

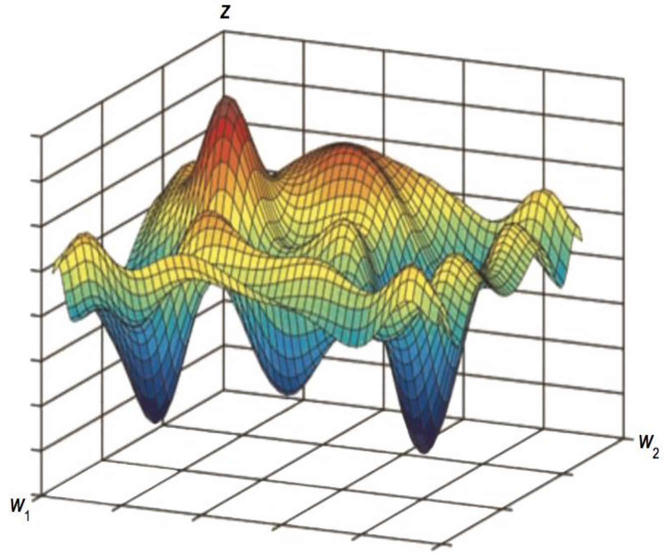
RNNs as dynamical systems



RNN state

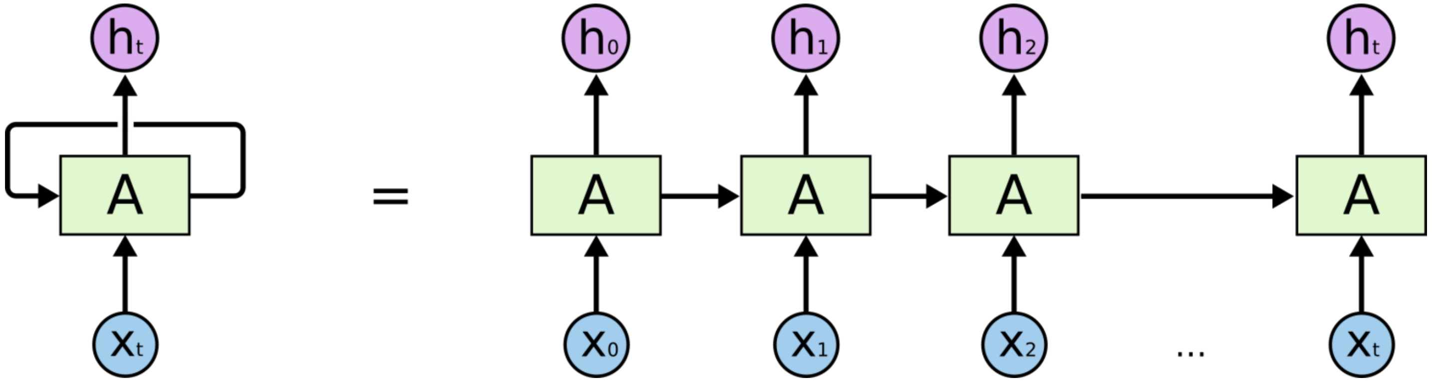
$$\boldsymbol{x} = \begin{pmatrix} x^1 \\ \vdots \\ x^n \end{pmatrix}$$

- An input vector $\boldsymbol{x_0} \in \mathbb{R}^n$ represents an initial condition for the nodes
- The nodes can take discrete or continuous values and update according to the RNN weights
  - Many possible update schemes
  - View state as particle in $\mathbb{R}^n$ whose evolution defines a trajectory $\boldsymbol{x}(t) \in \mathbb{R}^n$
  - The time steps can alternatively be discrete $\{\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_t, \dots\}$
- Well-behaved (non-diverging) RNNs converge to "attractors"
  - The set of initial points which converge to a given attractor defines its "basins of attraction"
  - The set of basins of attraction partition the state space $\mathbb{R}^n$
- The attractors can be simple fixed points, more complicated (e.g. limit cycle), or chaotic



Attractor landscape concept for fixed point RNNs

Viewing RNNs as FFNNs



- Unfolding view is well defined for deterministic fixed-point attractor networks
  - Architecture: "infinitely deep" layers which are all identical
  - Fixed-point convergence can be approximated with finitely many layers (i.e. truncate)
  - However, not well defined for e.g. chaotic dynamical systems

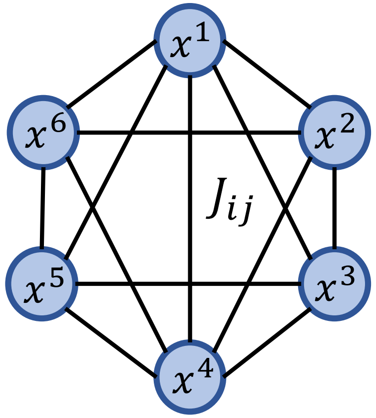
- There is an alternative FFNN interpretation of general non-divergent RNNs
  - Corresponding FFNN is one which partitions the state space into basins of attraction
  - The hidden layers in this case need not be identical – train a "vanilla" FFNN to learn basins
  - Didn't check if this is in the literature -- does it have any advantage over unfolding view?
  - Note the correspondence goes both ways (FFNN can be viewed as attractor RNNs)

http://colah.github.io/

Examples of RNNs

"Pure" RNNs + variants	Specialized RNNs + variants
Continuous Time Sigmoidal Networks (CTSN) *Use: Reproduce a deterministic dynamical system* **Boltzmann Machine (BM)** *Use: Model probability distributions, such as steady state of stochastic dyn. sys.* • Ising Model • Hopfield network • Restricted Boltzmann Machine (RBM) • Deep belief network (stacked RBMs)	**Long Short-Term Memory (LSTM)** *Use: Learn sequential data (e.g. text)* **Gated Recurrent Units (GRUs)** • Analogous to the LSTM unit, but with fewer parameters **Niche RNNs** **Reservoir Computing** • Echo-state network • Liquid state machine

Boltzmann Machines



- Same as Ising model when the nodes are boolean: $x^i \in \{+1, -1\}$
  - The edges J_{ij} are symmetric
  - The dynamics are generally *stochastic*

- BMs fall under "energy based models" in the ML literature
  - The dynamics have a Lyapunov function constructed from a Hamiltonian

$$H(\boldsymbol{x}) = -\frac{1}{2}\boldsymbol{x}^T\boldsymbol{J}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$$

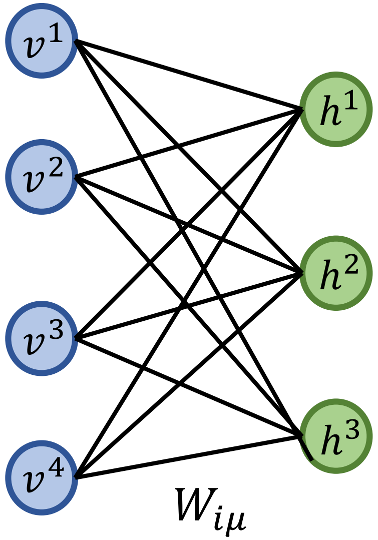
- Prototypical problem:
  - Given M samples $\{\boldsymbol{x}_i\}$ from the steady state distribution $p_{data}(\boldsymbol{x})$
  - Find $\boldsymbol{\theta} = \{\boldsymbol{J}, \boldsymbol{b}\}$ which maximize

$$L = \sum_i \ln \mathrm{p_{BM}}(\boldsymbol{x}_i|\boldsymbol{\theta})$$

- Example uses
  - Stochastic model reconstruction (e.g. "learning a Hamiltonian")
  - Generative modelling

- Issue: training is difficult
  - Computing $p_{BM}(\boldsymbol{x}|\boldsymbol{\theta})$ is expensive
  - Involves huge sum (2^n terms) $Z = \sum e^{-H(x)}$ at each training step

Restricted Boltzmann Machines



- Training RBMs is much faster
  - The hidden nodes $\boldsymbol{h}$ are independent given the visible nodes $\boldsymbol{v}$ (vice-versa)
  - This dramatically reduces the number of terms needed at each step of ML gradient ascent
  - ML approximations used in practice, e.g. (Contrastive divergence, Hinton, 2002)

- Dynamics: sequential (parallelized) updates of each layer

- 4 main RBM sub-classes:
  - $\boldsymbol{v}$ boolean, $\boldsymbol{h}$ boolean (most common)
  - $\boldsymbol{v}$ boolean, $\boldsymbol{h}$ cts (equiv. to Hopfield BM)
  - $\boldsymbol{v}$ cts, $\boldsymbol{h}$ boolean
  - $\boldsymbol{v}$ cts, $\boldsymbol{h}$ cts ("unstable"? –Hinton 2012)

- Possible to exactly transform BM to RBM using a Hubbard-Stratonovich transformation

- RBMs are the building block for "Deep belief networks"
  - Greedily trained stack of RBMs
  - Describe hierarchical features in data
  - Hinton, 2006

- RBMs are universal approximators of discrete distributions
  - Bengio, 2007

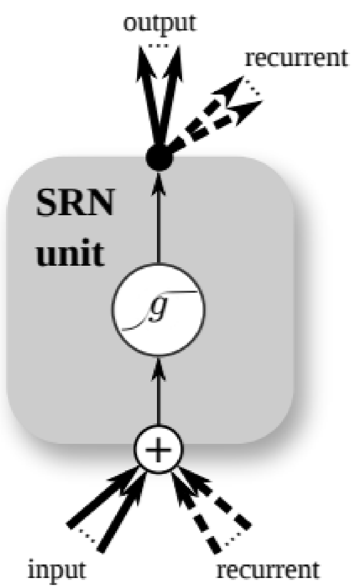
Long short-term memory (LSTM) networks

- Historic difficulties in training generic RNNs, especially for sequential input data
  - Standard training uses backprop after unfolding the RNN to an approximate, finite FFNN
  - Two common, separate problems: Gradient can vanish or explode during training
  - See [Bengio et al. 2013. On the difficulty of training Recurrent Neural Networks.]

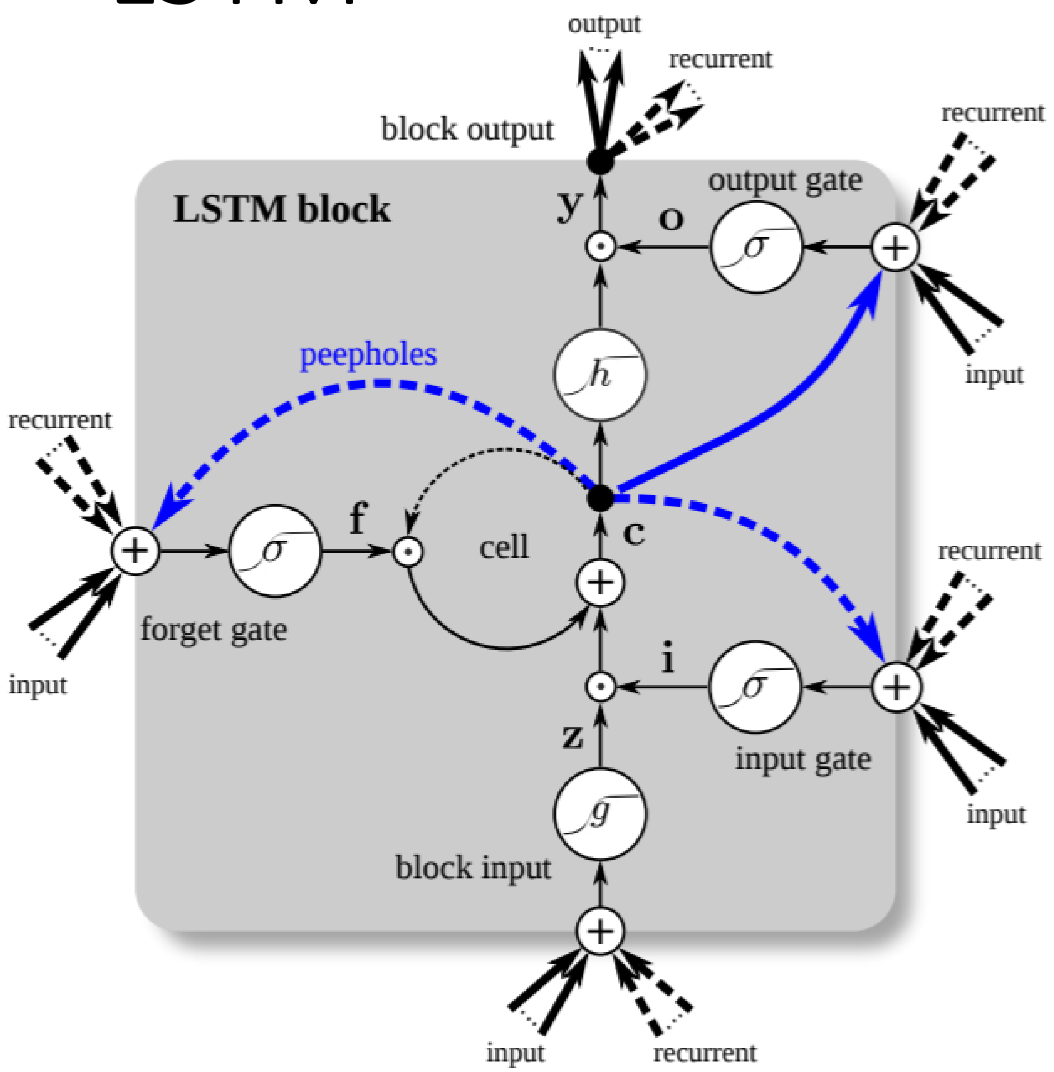
- LSTM developed to better apply RNNs to sequences.
  - Heuristically targets the vanishing gradient problem in generic RNN training
  - Original ref. Schmidhuber et al., 1997 (many variants since then)

- LSTM architecture appears convoluted, but is one of the most successful building blocks for sequence prediction, generation

Vanilla RNN



LSTM



Let $\mathbf{x}^t$ be the input vector at time t, N be the number of LSTM blocks and M the number of inputs. Then we get the following weights for an LSTM layer:

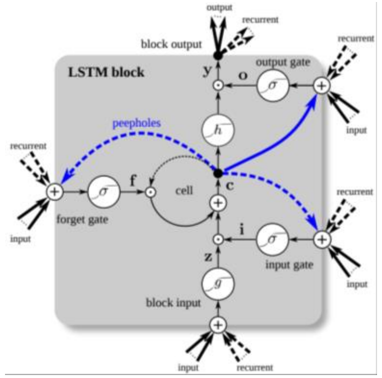
- Input weights: $\mathbf{W}_z, \mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o \in \mathbb{R}^{N\times M}$
- Recurrent weights: $\mathbf{R}_z, \mathbf{R}_i, \mathbf{R}_f, \mathbf{R}_o \in \mathbb{R}^{N\times N}$
- Peephole weights: $\mathbf{p}_i, \mathbf{p}_f, \mathbf{p}_o \in \mathbb{R}^{N}$
- Bias weights: $\mathbf{b}_z, \mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{N}$

Then the vector formulas for a vanilla LSTM layer forward pass can be written as:

$$
\begin{aligned}
\bar{\mathbf{z}}^t &= \mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z \\
\mathbf{z}^t &= g(\bar{\mathbf{z}}^t) && \textit{block input} \\
\bar{\mathbf{i}}^t &= \mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i \\
\mathbf{i}^t &= \sigma(\bar{\mathbf{i}}^t) && \textit{input gate} \\
\bar{\mathbf{f}}^t &= \mathbf{W}_f\mathbf{x}^t + \mathbf{R}_f\mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f \\
\mathbf{f}^t &= \sigma(\bar{\mathbf{f}}^t) && \textit{forget gate} \\
\mathbf{c}^t &= \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t && \textit{cell} \\
\bar{\mathbf{o}}^t &= \mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^{t} + \mathbf{b}_o \\
\mathbf{o}^t &= \sigma(\bar{\mathbf{o}}^t) && \textit{output gate} \\
\mathbf{y}^t &= h(\mathbf{c}^t) \odot \mathbf{o}^t && \textit{block output}
\end{aligned}
$$

Schmidhuber et al., 2017. LSTM: A Search Space Odyssey.

Re-drawing the figure…



Let $\mathbf{x}^t$ be the input vector at time t, N be the number of LSTM blocks and M the number of inputs. Then we get the following weights for an LSTM layer:

- Input weights: $\mathbf{W}_z$, $\mathbf{W}_i$, $\mathbf{W}_f$, $\mathbf{W}_o \in \mathbb{R}^{N \times M}$
- Recurrent weights: $\mathbf{R}_z$, $\mathbf{R}_i$, $\mathbf{R}_f$, $\mathbf{R}_o \in \mathbb{R}^{N \times N}$
- Peephole weights: $\mathbf{p}_i$, $\mathbf{p}_f$, $\mathbf{p}_o \in \mathbb{R}^N$
- Bias weights: $\mathbf{b}_z$, $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o \in \mathbb{R}^N$

Current timestep

Previous timestep

Input vector

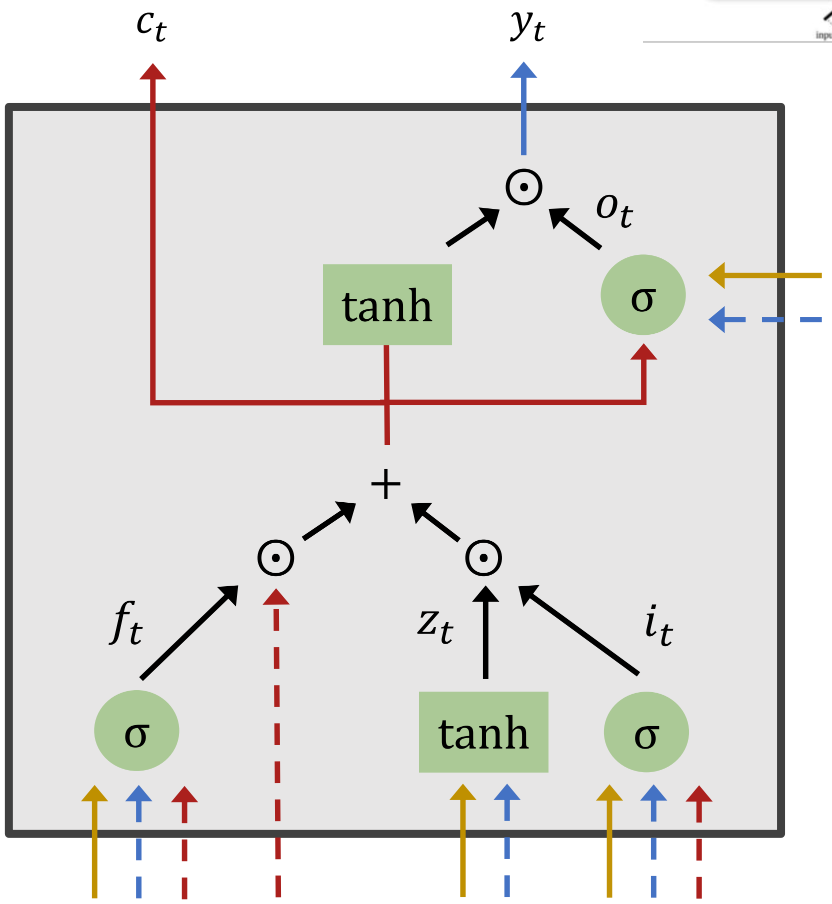
x_t

RNN state vector

y_{t-1}

RNN "cell" vector

c_{t-1}



Dynamics

Four LSTM "Gates"

$$z_t = \tanh(W^z x_t + R^z y_{t-1})$$
$$i_t = \sigma(W^i x_t + R^i y_{t-1} + p^i \odot c_{t-1})$$
$$f_t = \sigma(W^f x_t + R^f y_{t-1} + p^f \odot c_{t-1})$$
$$o_t = \sigma(W^o x_t + R^o y_{t-1} + p^o \odot c_t)$$

"Cell state"

$$c_t = z_t \odot i_t + f_t \odot c_{t-1}$$

"RNN state"

$$y_t = \tanh(c_t) \odot o_t$$

Remarks

- LSTMs are often said to “perform better” than generic RNNs for “most tasks”

- However, recall the dynamical system approximation theorem for RNNs.
  Denote the LSTM state as $\boldsymbol{x} = [\boldsymbol{y}, \boldsymbol{c}]$. Since LSTM is a dynamical system, there exists $\boldsymbol{W}, \boldsymbol{h}, \boldsymbol{\tau}$ such that its continuous time evolution is approximated by

$$\boldsymbol{\tau} \frac{d\boldsymbol{x}}{dt} = -\boldsymbol{x} + \boldsymbol{W}\sigma(\boldsymbol{x}) + \boldsymbol{h}$$

- Important note: The above only works in the absence of input (I think)

- Extra: [von Brecht, Laurent, 2016] claim to show that LSTM (and GRU) input-less dynamics follow chaotic attractors, and propose an analogous but stable architecture

Some RNN use cases

- “I want to model a deterministic dynamical system”
  - Use a continuous time RNN
  - Generative use: believable trajectories

- “I want to model a probability distribution / extract features”
  - Use an RBM or Deep RBM
  - Generative use: believable samples

- “I want to model sequential data (e.g. text, speech)”
  - Use an LSTM / LSTM variant
  - Generative use: believable text, music, etc.

Some Refs

- BM and RBM
  - Hinton. 2012. A Practical Guide to Training Restricted Boltzmann Machines.
  - Salakhutdinov, Hinton. 2009. Deep Boltzmann Machines.
  - Le Roux, Bengio. 2007. Representational Power of Restricted Boltzmann Machines and Deep Belief Networks.
  - Bengio, Simard, Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult.
  - Bengio et al. 2013. On the difficulty of training recurrent neural networks.

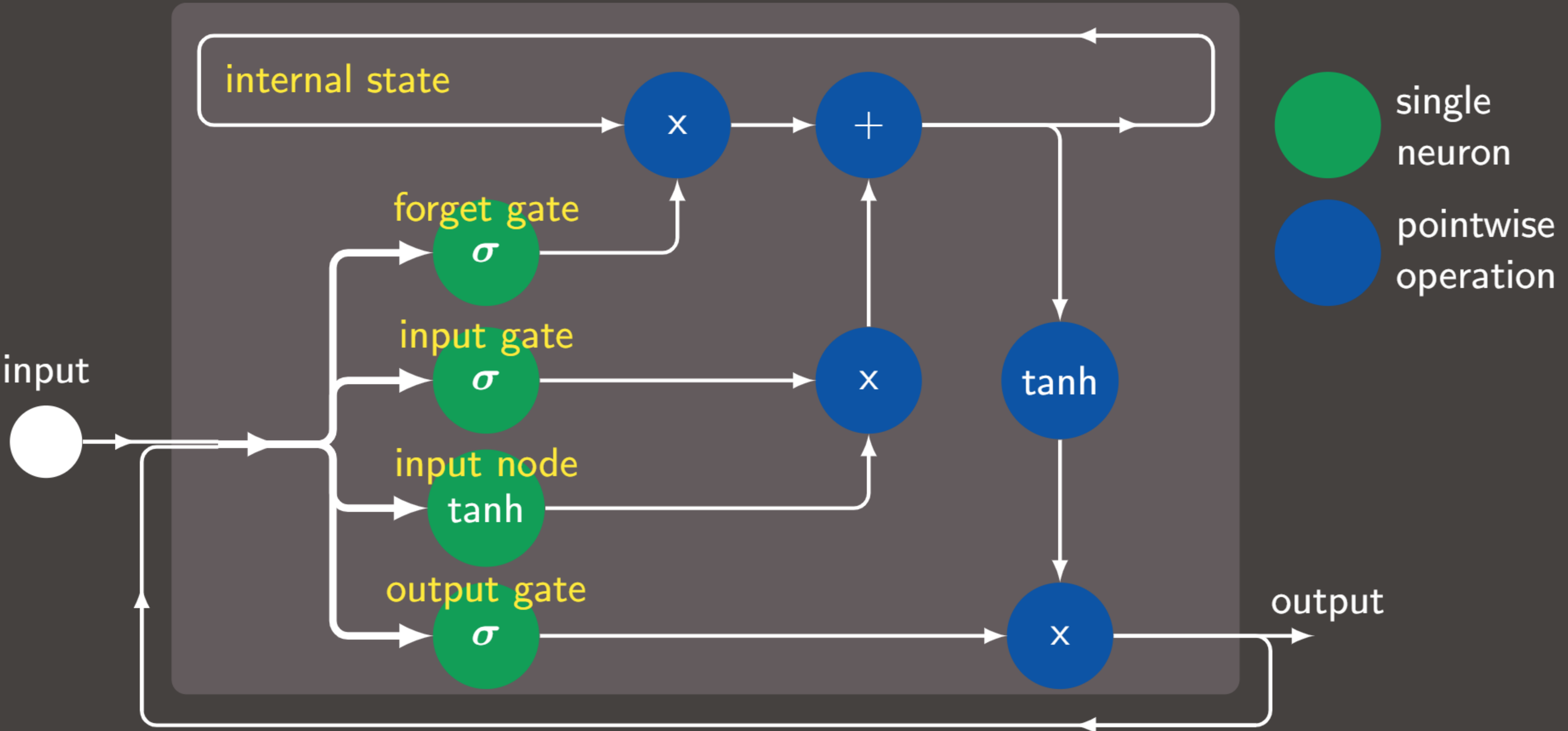
- LSTM
  - Schmidhuber site: http://people.idsia.ch/~juergen/lstm/
  - Google blog: http://colah.github.io/posts/2015-08-Understanding-LSTMs/
  - Schmidhuber et al. 2017. LSTM: A Search Space Odyssey.

- Misc
  - Funahashi, Nakamura. 1992. Approximation of dynamical systems by continuous time recurrent neural networks.
  - Laurent, Brecht. 2016. A recurrent neural network without chaos

- Reviews
  - Mehta et al. 2018. A high-bias, low-variance introduction to Machine Learning for physicists.
  - Bengio, Courville, Vincent. 2014. Representation Learning: A Review and New Perspectives.
  - Tanaka et al. 2019. Recent advances in physical reservoir computing: A review.

SciNet LSTM Slides

Long Short Term Memory networks, memory cells



Notes about LSTM memory cells

Some notes about these memory cells.

- The 'input node' is a standard input node. These typically use a tanh activation function, though others can be used.
- How much of the input is added to the 'internal state' is controlled by the 'input gate.'
- The 'forget gate' controls how much of the internal state we're keeping, based on the input.
- The 'output gate' controls how much of the internal state is output.
- The internal state is put through a tanh function before output. This is optional, and is only done to put the output in the same range (-1 to 1) as a typical hidden layer. Some implementations use other functions such as rectifier linear units.

Notes about LSTMs

Some notes about LSTMs in networks.

- Each 'memory cell' is treated like a single neuron in a hidden layer. Typically there are many such cells in such a layer.
- In the Keras implementation of LSTMs, not only is the output of a single LSTM cell concatenated to its input, the output of all the LSTM cells in the layer are concatenated to the input.
- These networks are trained in the usual way, using Stochastic Gradient Descent and Backpropagation, as with other neural networks.
- These have been used in language translation, voice recognition, handwriting analysis, next-letter prediction, and many many other applications.

LSTM example

One common application of LSTMs is text prediction. Let's use an LSTM network to create a recipe.

- We will use the recipe data set, which is a text file containing 4869 recipes.
- We take the recipe data set, as a single file, and analyse it to find all unique words.
- We then one-hot-encode the words in the data set using our word list.
- We then break the data set into 50-word one-hot-encoded chunks ("sentences").
- We will then train the network:
  - the input will be the 50-word-encoded chunks.
  - the target will be the next word in the data set.
- Once the network is trained we can feed the network a random sentence as a seed, and it will use that sentence to generate new words, until we have a new recipe.

One-hot encoding

One way of portraying sentences is one-hot encoding. In this representation, all words are given an index in a vector of length num_words. The word gets a '1' when the word occurs and a '0' when it doesn't. The sentence then consists of an array of sentence_length rows and num_words columns.

Consider the sentence "The dog is in the dog crate."

The number of unique words is 5. Each word gets its own index: {the: 0, dog: 1, is: 2, in: 3, crate: 4}.

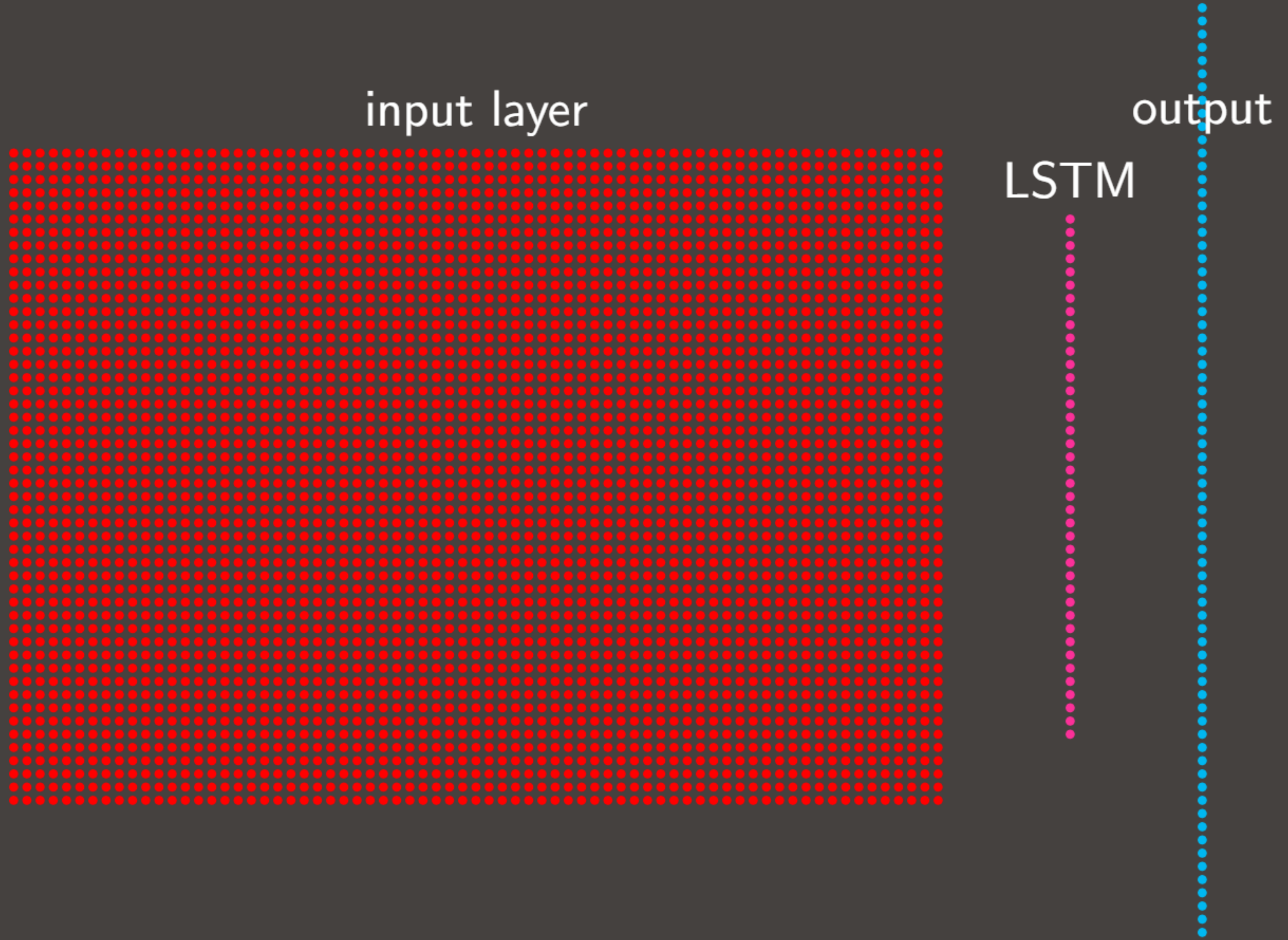
The sentence above can then be represented by the matrix to the right, with dimensions (sentence_length, num_words).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Our LSTM network

The network is simple:

- The input has dimensions (sentence_length, n_words)
- sentence_length = 50
- n_words = number of unique words in the data.
- The LSTM layer has 256 nodes.
- The output layer is fully-connected, of length n_words.



LSTM example, learning code, continued

```
# Learn_Recipes.py, continued

# Save the metadata.
g = shelve.open("data/recipes.shelve")
g["sentence_len"] = sentence_len
g["n_words"] = n_words
g["encoding"] = encoding
g["decoding"] = decoding
g.close()

# Create the NN.
model = km.Sequential()

# A layer of LSTMs.
model.add(kl.LSTM(256,
  input_shape = (sentence_len, n_words)))
```

```
# Add a fully-connected output layer.
model.add(kl.Dense(n_words, activation = 'softmax'))

# The usual compilation.
model.compile(loss = 'categorical_crossentropy',
  optimizer = 'sgd', metrics = ['accuracy'])

# Run the fit.
fit = model.fit(x, y, epochs = 200,
  batch_size = 128, verbose = 2)

# Save the model.
model.save('data/recipes.model.h5')
```