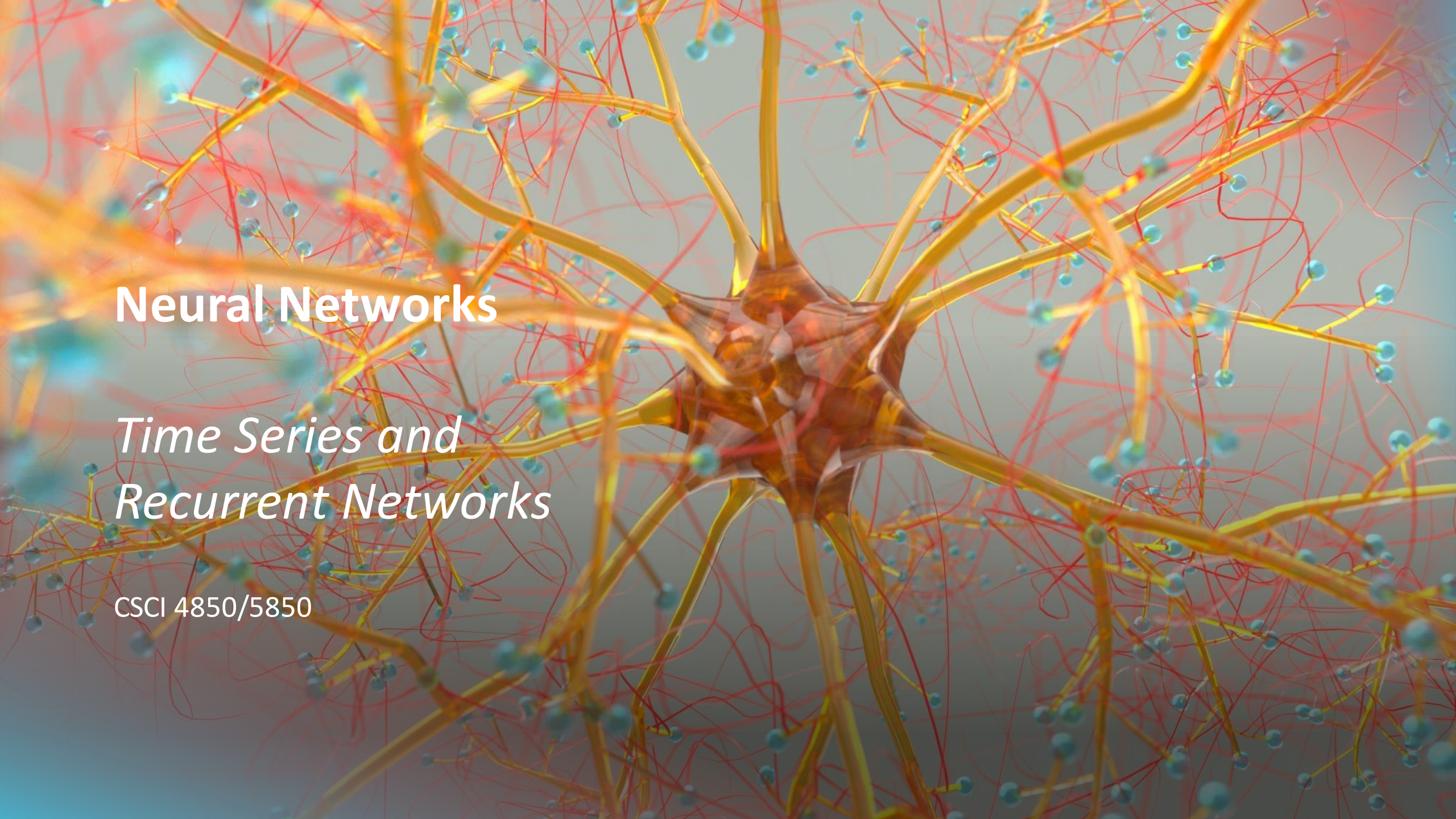


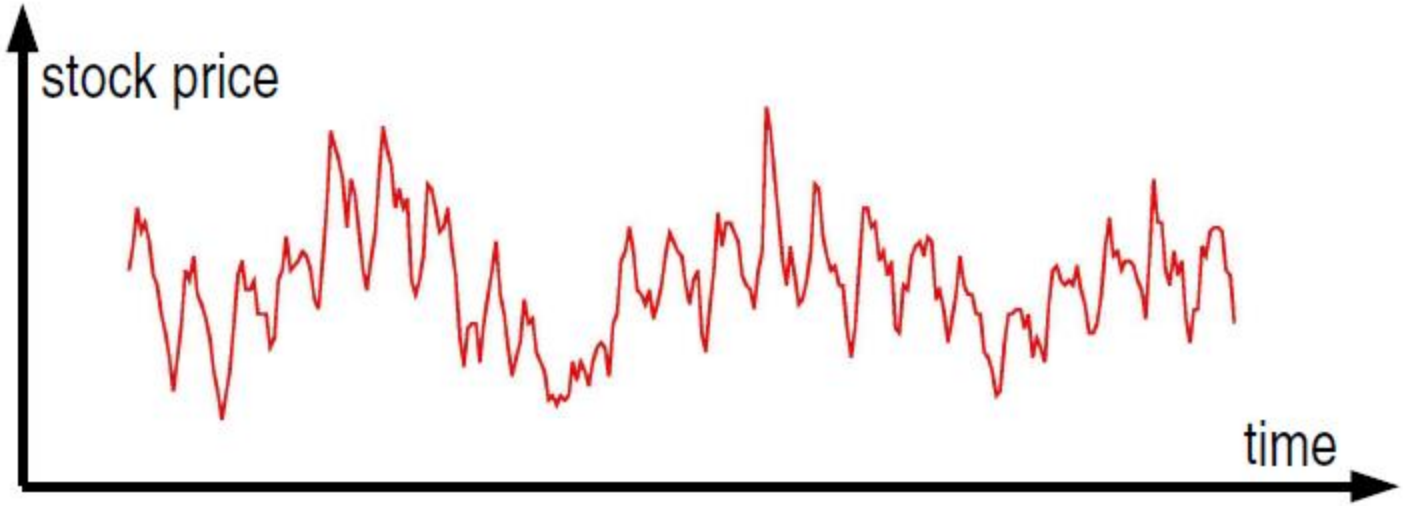
Neural Networks

Time Series and Recurrent Networks

CSCI 4850/5850

Learning to Predict the Future…

- Given information from the past, can the future be predicted (classification *or* regression)?
- Some data lend themselves naturally to this kind of problem interpretation..



Temporal Data

Audio – naturally temporal data

Classification of music/audio streams

Prediction or modeling of music/audio

Video – naturally temporal data

Classification of movies/video streams

Prediction or modeling of movies/video streams

Natural Language Processing – naturally temporal data

Classification of sentence/paragraph/document meaning

Prediction of modeling of sentence/paragraph/document construction

Granularity: audio (speech recognition), character (optical character recognition), word, low-dimensional projection in character/word/paragraph/document space…

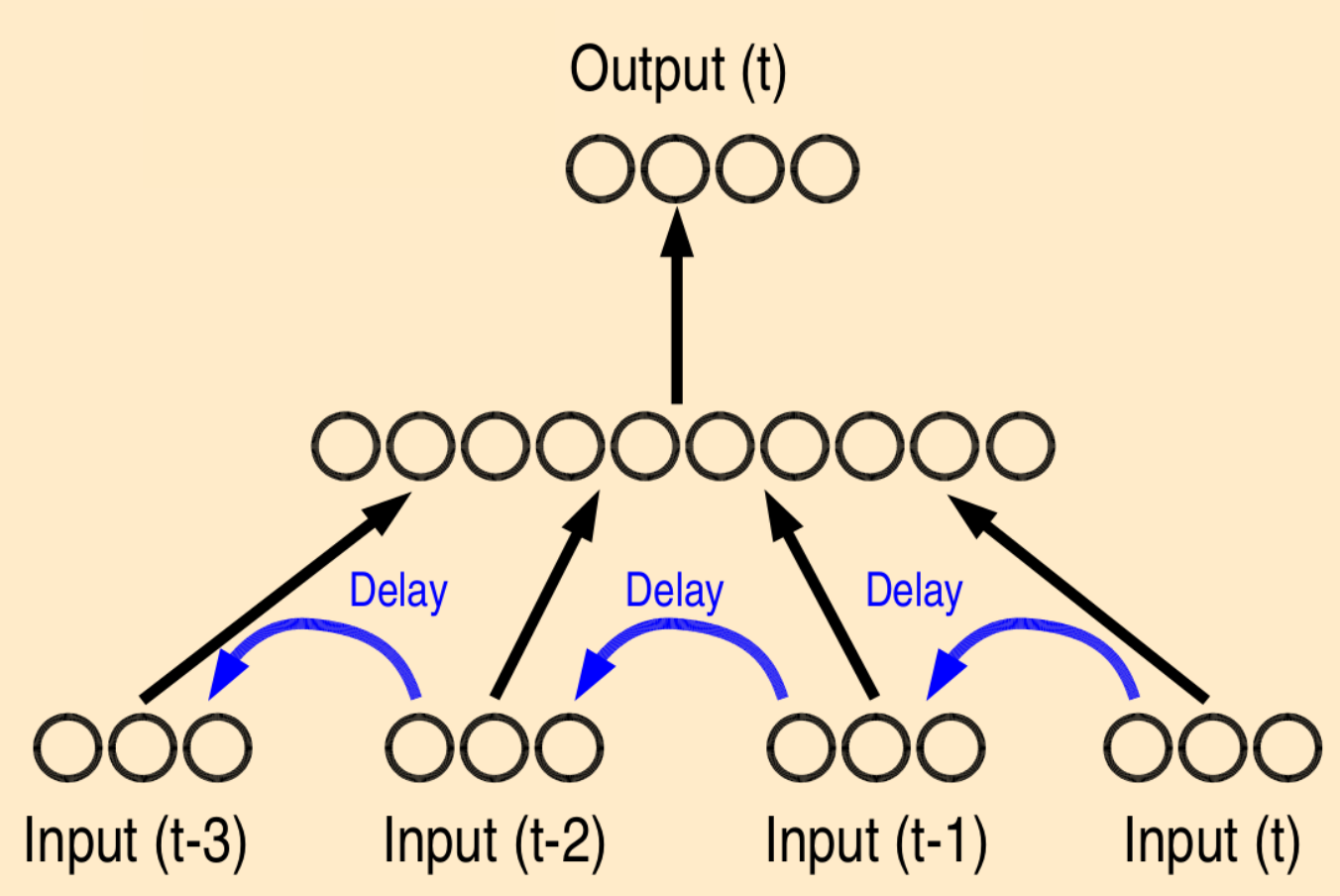
Weather, heart rhythms, brain waves, bird flocking, insect swarming, molecular motion, RNA expression patterns, etc.

Key Ideas

Some data cannot be analyzed using a *single measurement* alone (constrained in time)

Information from the *past* may need to be **remembered/stored** for making a classification/regression decision *now or in the future*

To store or not to store?… it's an important question!

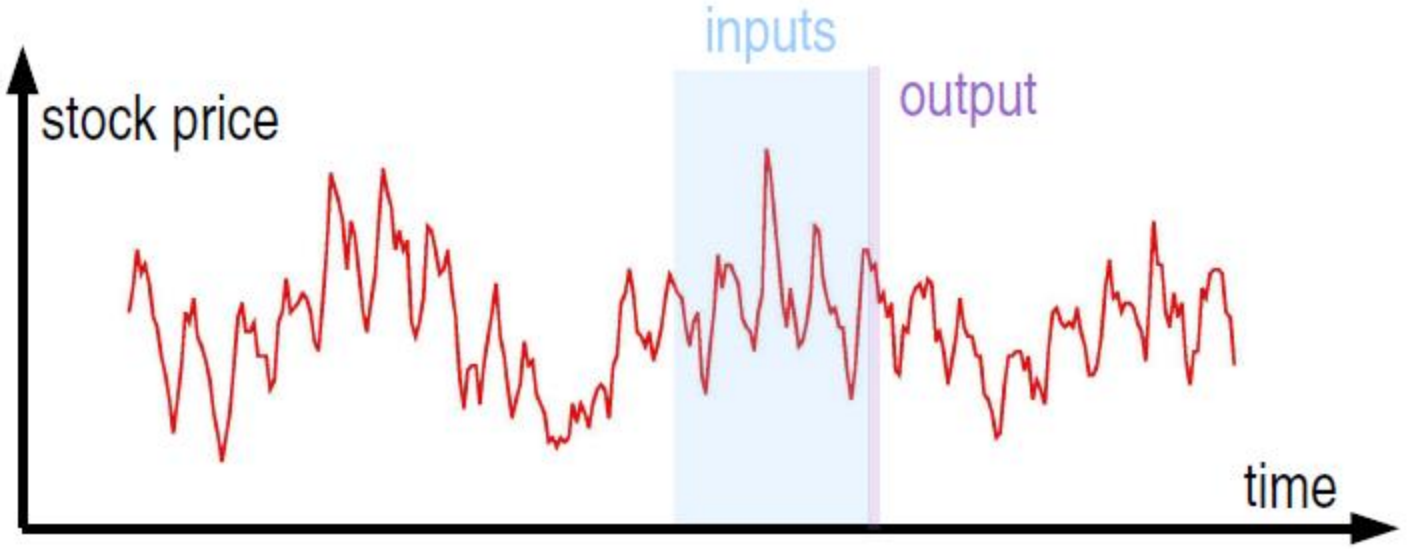


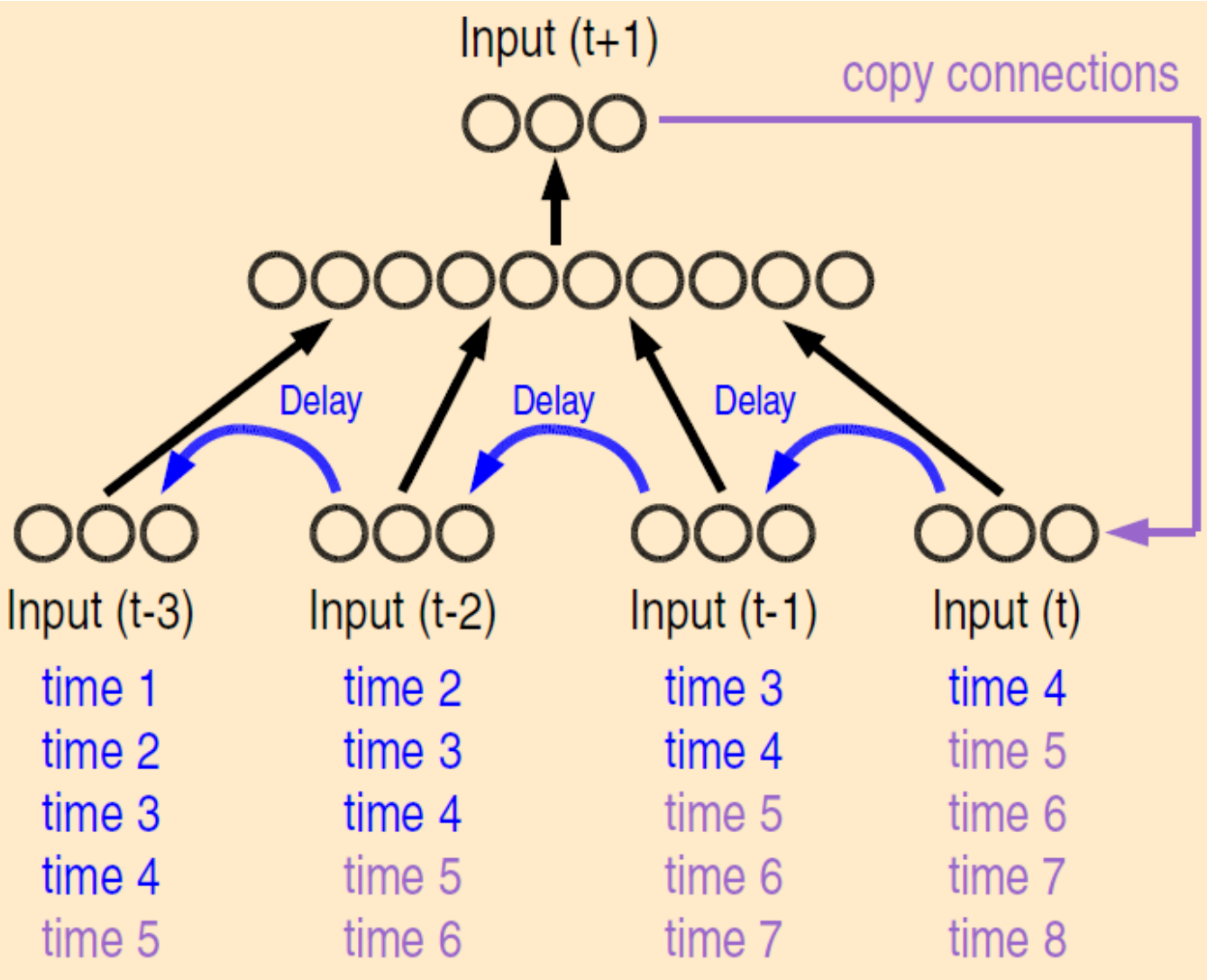
The Feed-forward Way

- Time series data consist of feature vectors from **discrete** time points
- Include **multiple** time points in the past as inputs…
- Unroll time in space…
- *Auto-regressive* model!
- Continuous extensions exist, but discretization is more common than continuous neural networks

Key Idea: The Markov Assumption

- We *must* fix the size of the input (number of time steps) – can't keep increasing the size of the network…
- We must therefore make the **Markov Assumption**:
  - The function we are learning (next state of the system) depends only on a *finite number* of past states/inputs (collectively assumed to be the previous state)





More than one step in the future?

- We could attempt to forecast more
- Prediction for longer stretches of time can be achieved by copying back
- We must therefore make the **Markov Assumption**:
  - The function we are learning (next state of the system) depends only on a *finite number* of past states/inputs (collectively assumed to be the previous state)

Potential Problems

All inputs are retained in the allotted temporal window, and therefore, all inputs are considered when performing the task…

Are some inputs more *relevant* than others?

Is this how the brain works with temporal information?

Did we pick the right time-frame?

Too short, can't solve the task

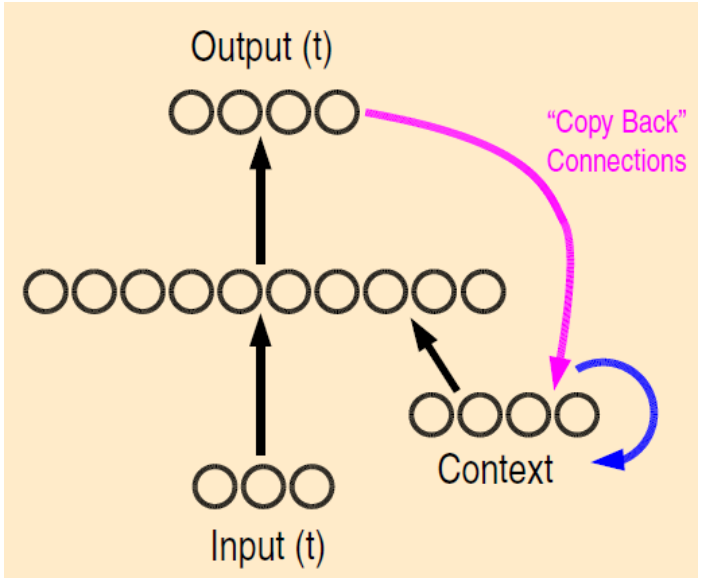
Too long, *distracting* inputs may impede learning

Relative vs. Absolute position in time…

Similar patterns [000110], [011000] don't look similar!

Beyond Feed-forward…

- Feedforward – combinational circuits
  - Inputs -> Outputs
- Feedback – sequential circuits
  - Inputs + Current Output -> Next Output
- Jordan, 1986
  - So-called Jordan Networks
- Simple Recurrent Network



Jordan Networks

Success

- First neural networks with consistent, stable sequence learning
- Used standard learning techniques (backprop!)

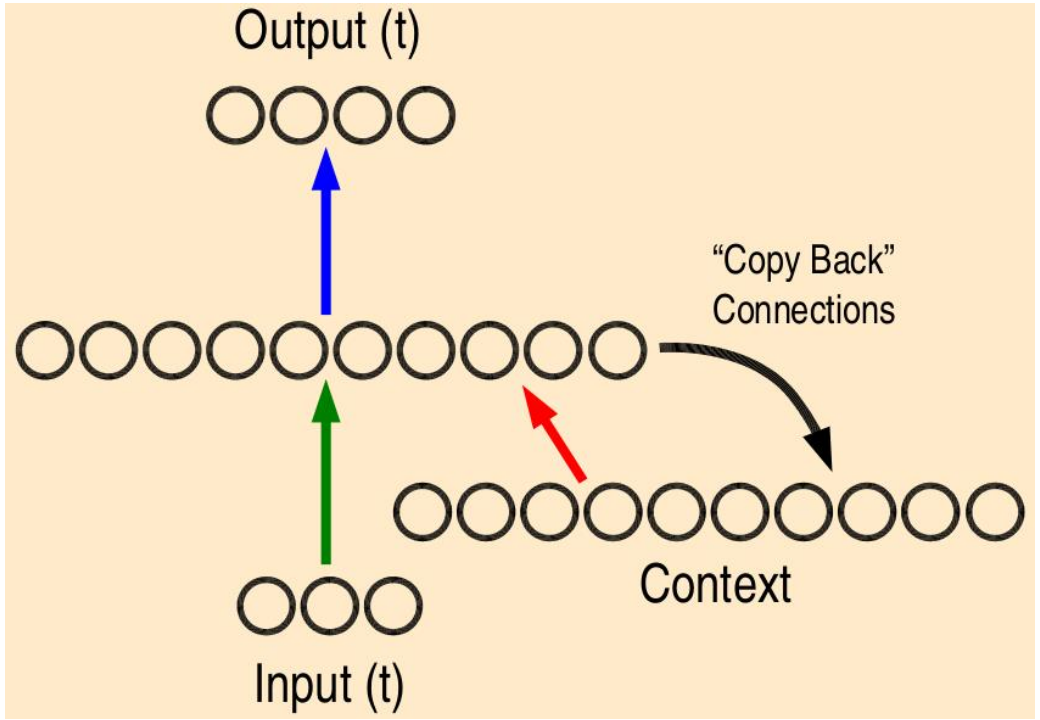
Limitation

- The **context** layer holds a *low-dimensional* representation (classification/regression) that needs to be “solved again” by the network during the next forward pass

Elman Networks

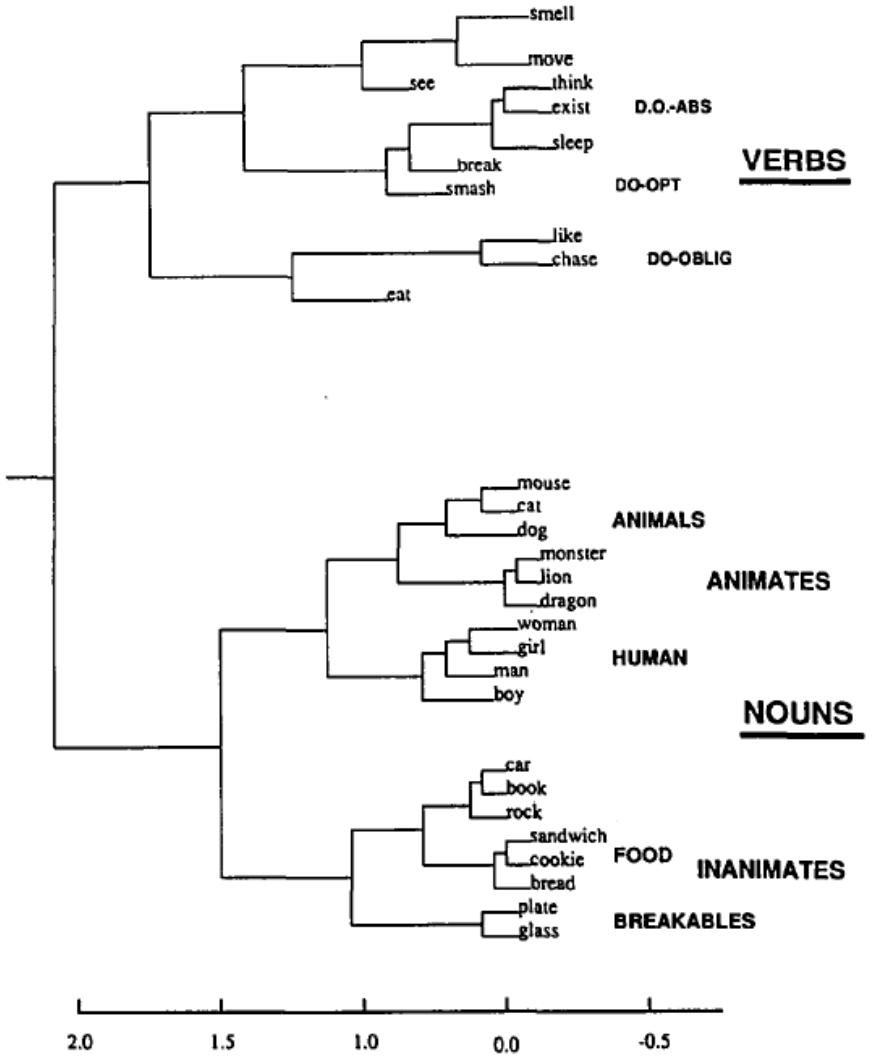
- Store the hidden representation instead…
- Elman, 1990

- Simple Recurrent Network
  - Again, traditional learning rules apply



Elman Networks

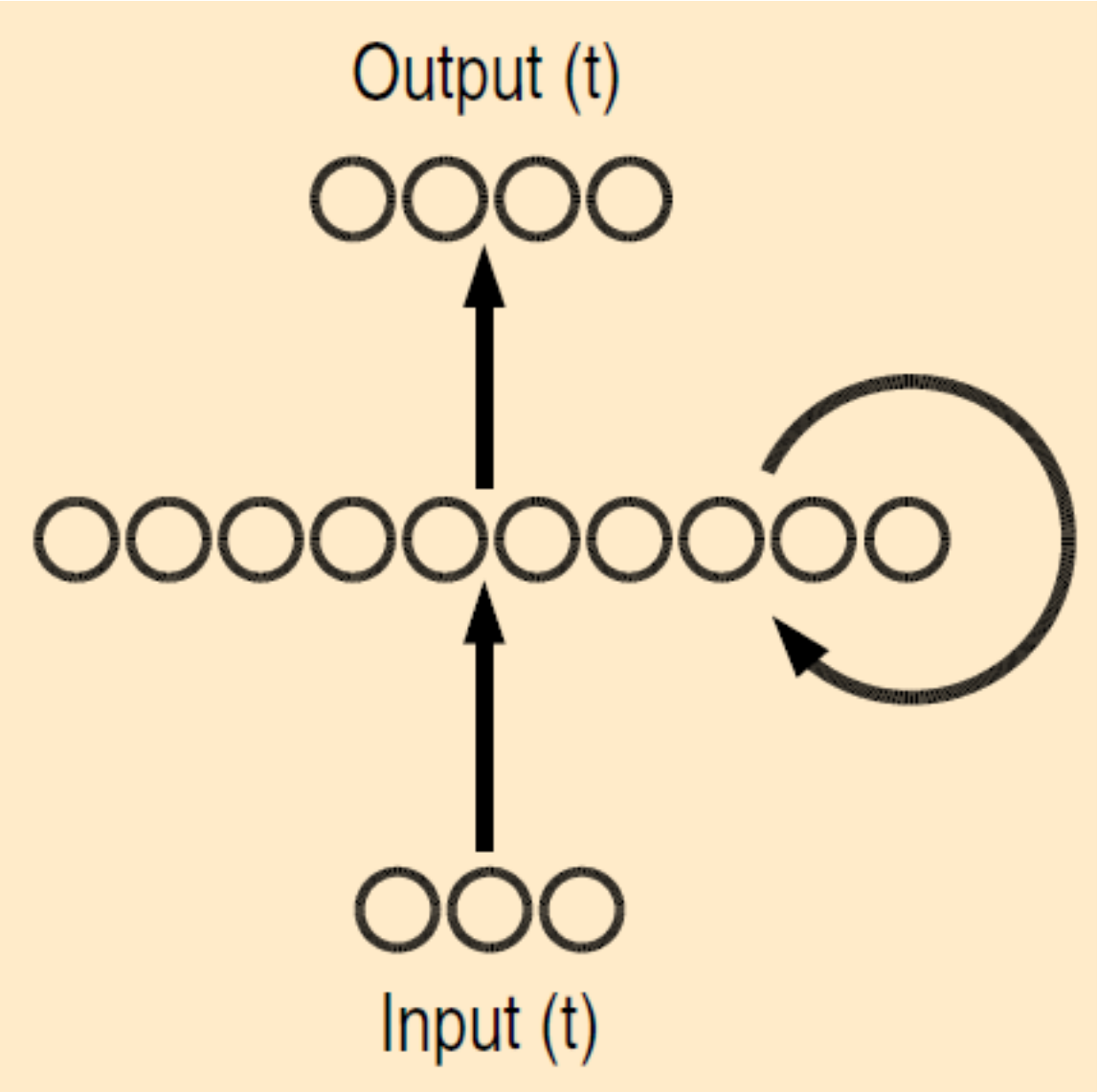
- Success
  - Hidden layer representations showed "hierarchical structure"
- Limitation
  - Traditional learning rules can't adapt hidden unit **copy-back** connections
  - The temporal *invariance* of representations might not be learned…



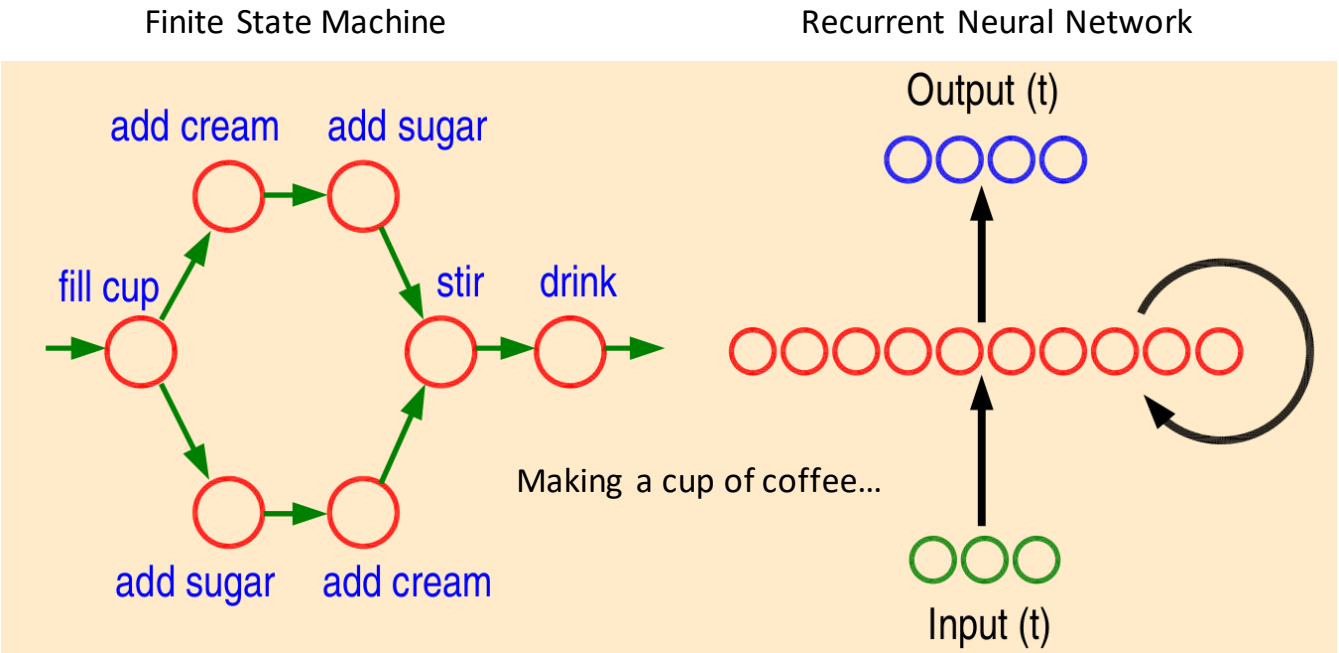
Clustering of hidden unit activation vectors

Moving Beyond SRNs

- The **general** architectural form of a recurrent neural network seems straightforward…
- What do these networks learn *in general*?
- What techniques allow such networks to learn *in general*?



The "Graded State Machine" Analogy



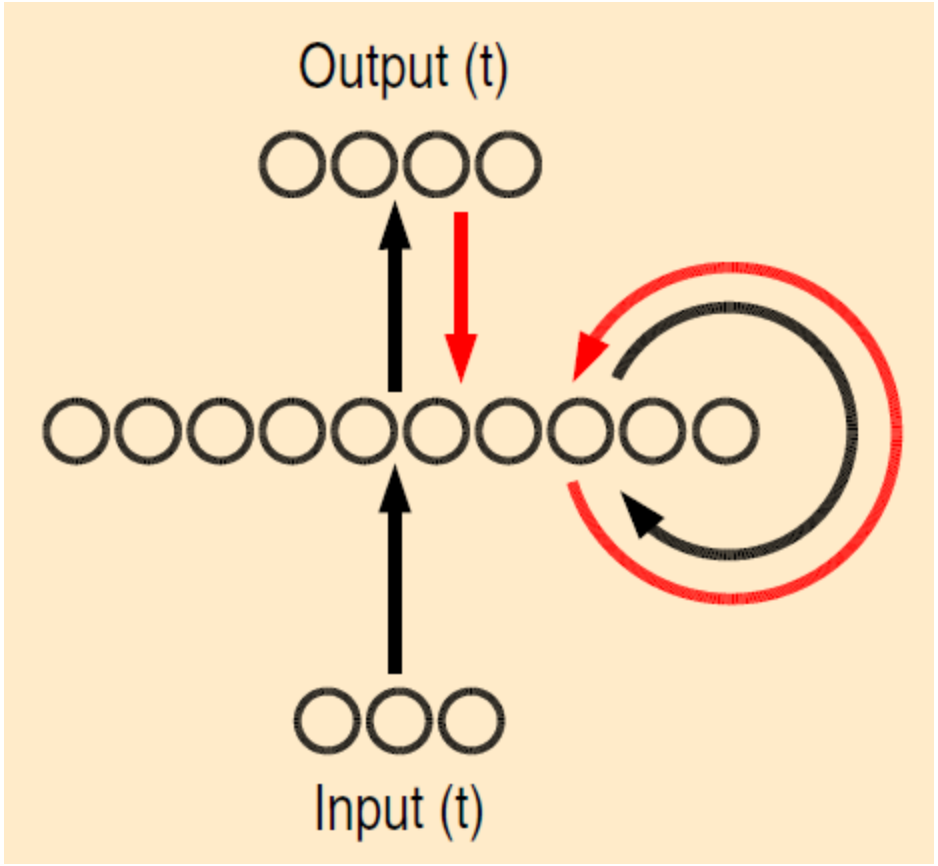
Feed-forward NNs are generalized *function approximators* [y=f(x)]

Recurrent NNs are generalized *state machines* [$y_t=f(x,y_{t-1})$]

(real-valued or smoothed state transitions are even possible!)

Backpropagation of Error?

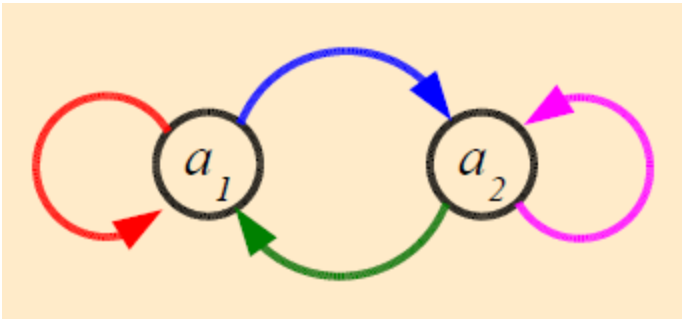
- Hidden to Output weight updates can be computed as expected…
- What do we handle the deltas for the hidden units?

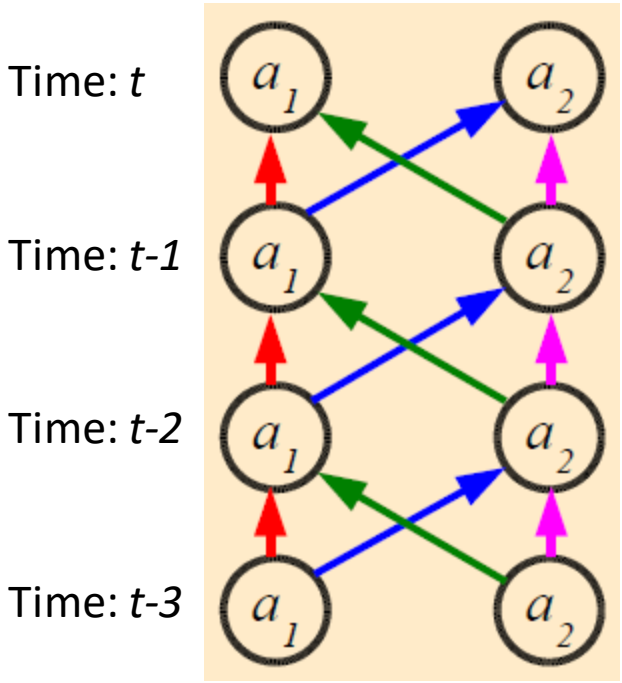


Backpropagation Through Time (BPTT)

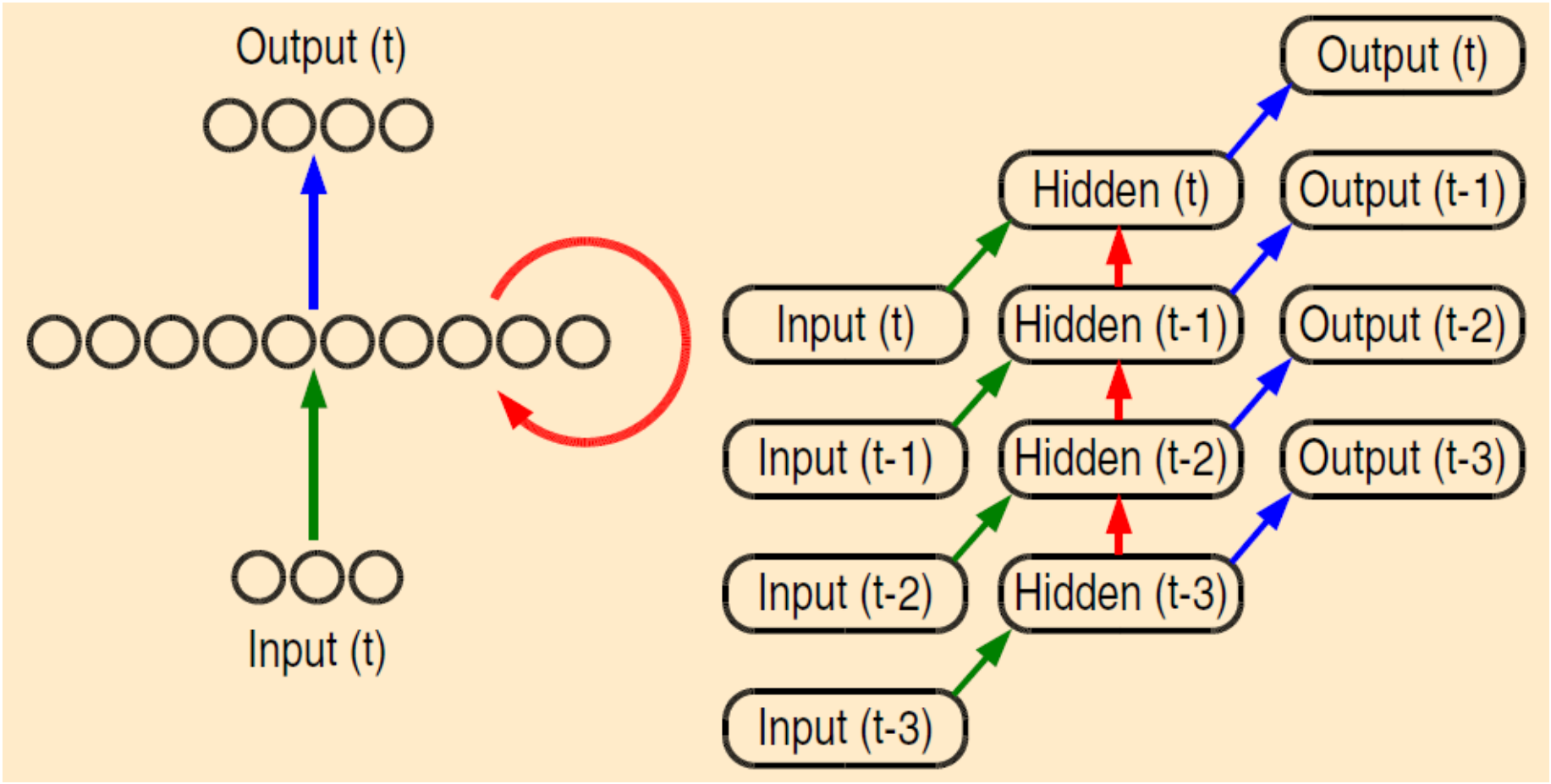
- Unroll *time in space*

- In this way, any connection can be updated *at each moment in time*
- Need to keep careful track of different *deltas and activations* which occur on different timesteps…





Unrolling a Hidden Layer

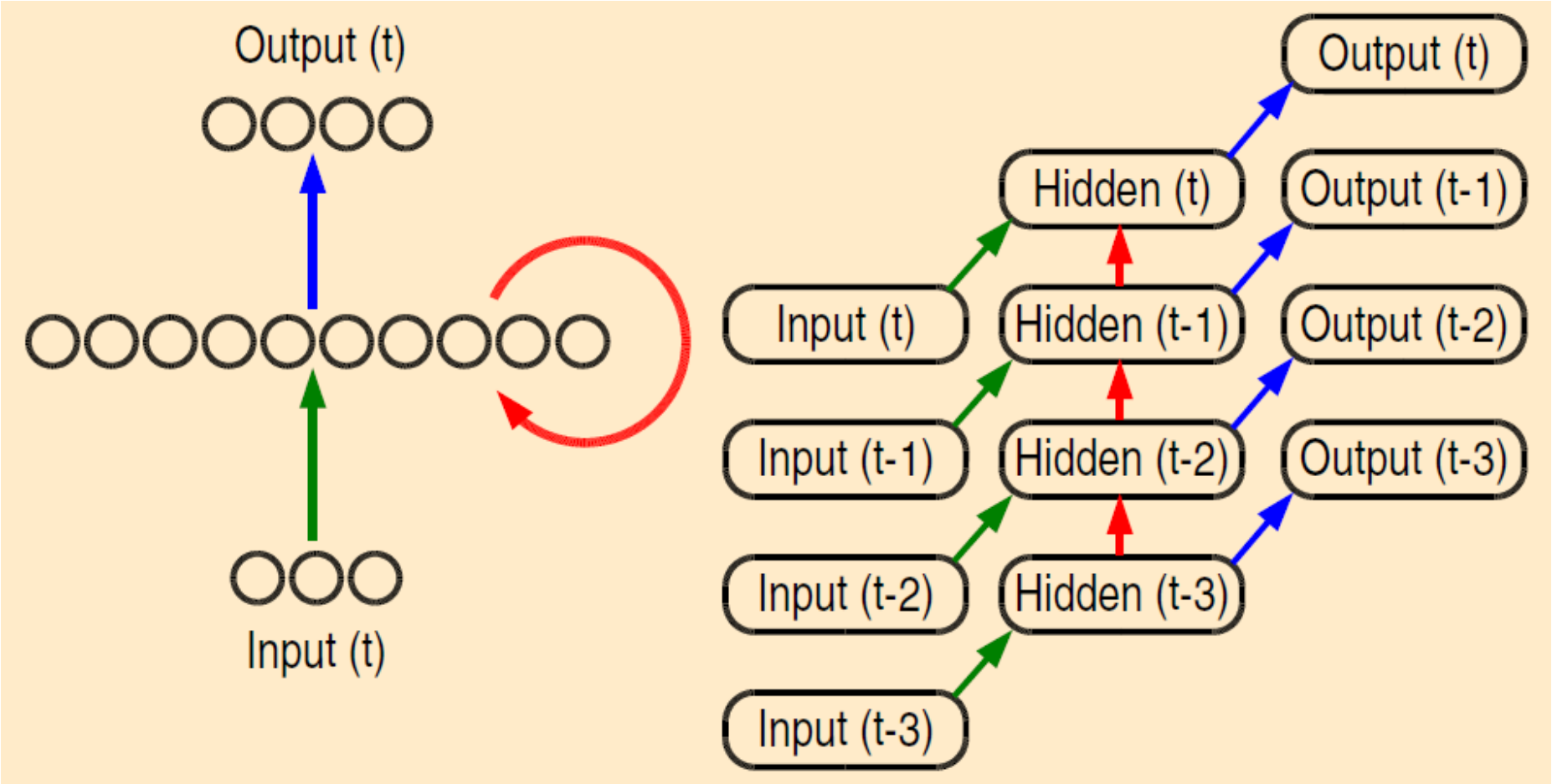


When to map?

- Different problems require different assumptions, and make some minor changes to the architecture of recurrent nets…
  - One-to-Many (Picture to Description)
  - Many-to-Many (Stocks, Audio, Video, NLP)
  - Many-to-One (Audio, Video, etc.)
- The typical problems of feed-forward networks apply for recurrent neural networks
  - Gradient optimization
  - Non-linear decision boundaries/functions
  - Overfitting/generalization performance

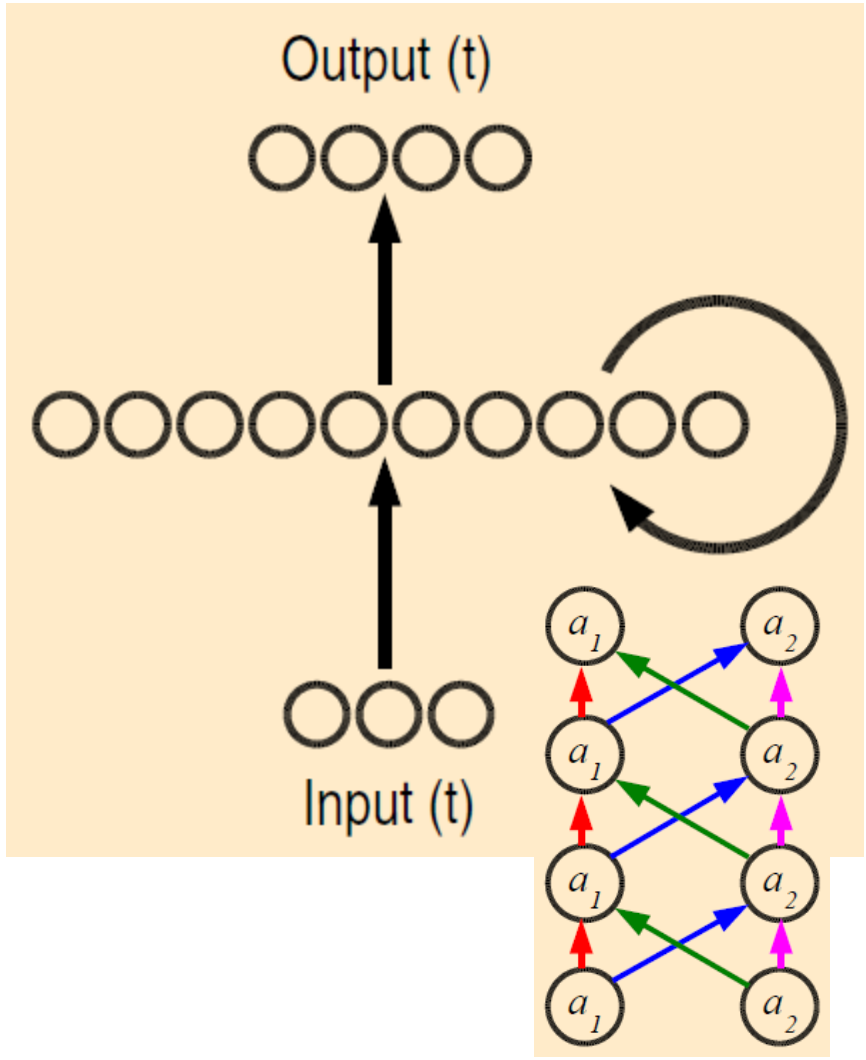
The Recurrent Neural Network

We often call this a "simple" recurrent network these days, but it uses BPTT!



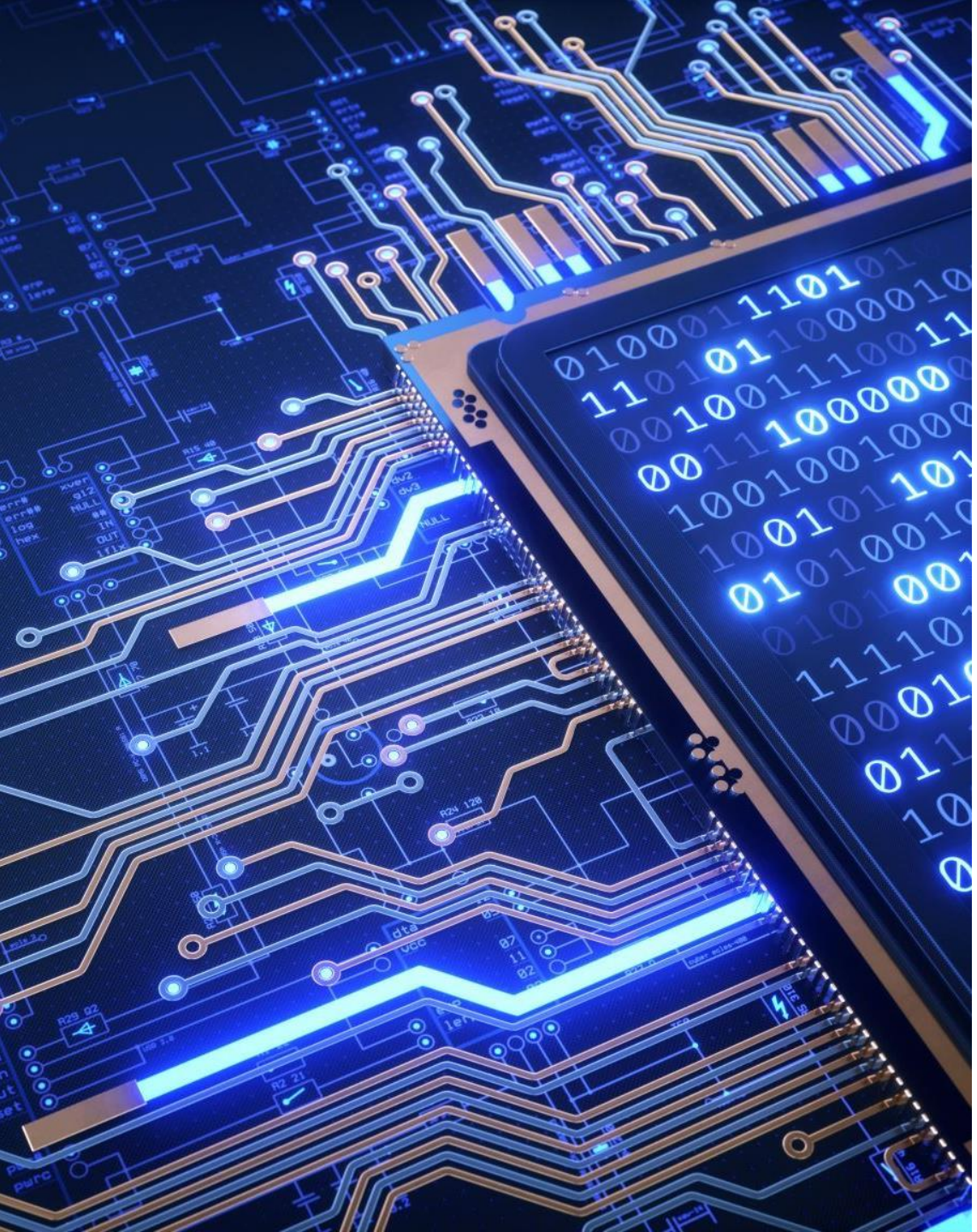
Remaining Issues with Simple RNNs

- Exploding gradients
  - Weights are often reused during the BPTT algorithm
  - Weights >1 make the gradient grow
  - Pushes units to saturation (slows learning/relearning)
- Shrinking gradients
  - Same rationale as above, but for weights <1
  - Pushes error signals away (no differentiation
- Every step counts!
  - Some information from –many-timesteps prior needs to be stored for proper prediction
  - The weights need to somehow "protect" this information until the necessary time
  - Also, some information may be needed for some time, but then is no longer needed and will continue to impact the processing of the network…



Looking forward…

- BPTT is *still* the tool of choice for training modern recurrent neural network (RNN) architectures
- However, some key insights from RNN research (and a little more inspiration from the brain) have led to the development of specialized network architectures for learning serialized data
- Again, tuning the ***inductive bias*** of the network for serialized data will be a useful technique, just as convolution was a useful technique for structured data…

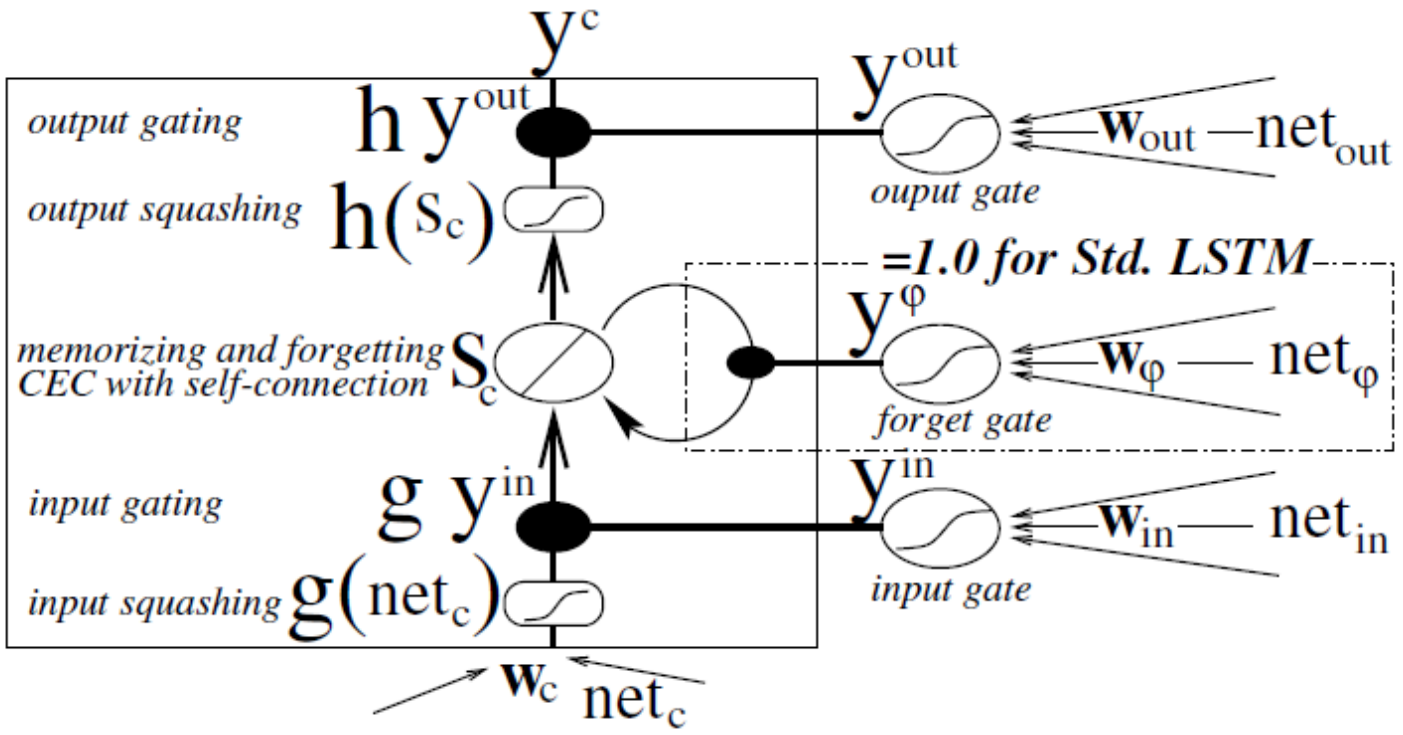


Long Short-term Memory (LSTM)

- Hochreiter and Schmidhuber (1997)
  - Dealt with the idea of appropriate storage and "protection" of past information
- Gers, Schmidhuber, and Cummins (2000)
  - Added support to "forget" information that was no longer needed in the future (could be folded into other units too, freeing up space)
- There are other recurrent layer architectures that have been proposed, but LSTM is the current *de-facto* standard for recurrent neural networks…
- For the most part, you can "drop-in" an LSTM layer of units similar to a SimpleRNN
  - Only modification is that the LSTM can produce two types of output (both a vector of remembered information, and an output mapping of that information transformed an activation function)

LSTM Architecture

- Significantly more complex than traditional neural units: LTSM Cell Architecture
- Each cell has ***four*** sets of weights which feed into it (instead of one)
- Output squashing can take any **activation function** we want though…
- Learns: 1. What/when to let something in, 2. When to forget, 3. What/when to let something out
- Surprisingly, there is some biologically plausible evidence for this although not at the "single cell" level, several neural circuits are involved (we will see this later…)

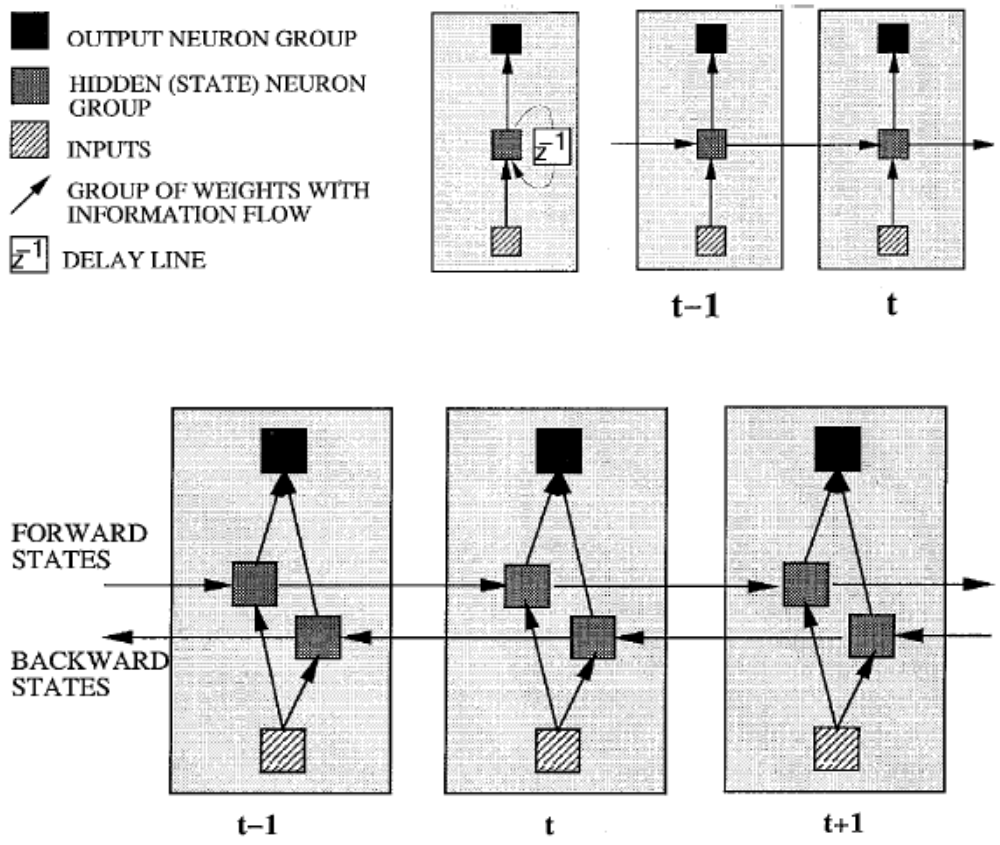


Still more to be done…

- LSTM *can* still result in gradient issues…
  - The frequency of these problems is reduced
  - The magnitude of these problems is reduced
  - Still, some temporal information needs to be folded across the same weights for several time steps, and may still cause issues…
- Gradient clipping and Regularization (Pascanu, et al. 2013)
  - Clip large gradients
  - Smooth vanishing gradients
  - *Balance* between both can sometimes be achieved

Alternatives: Bidirectional RNNs

- (Schuster and Paliwal, 1997) and more recent success by (Graves et al. 2013)
- Having access to information about the next time step helps training (can still use predictions to seed backward pass when needed – *inference*)



LSTM for General Sequence Learning

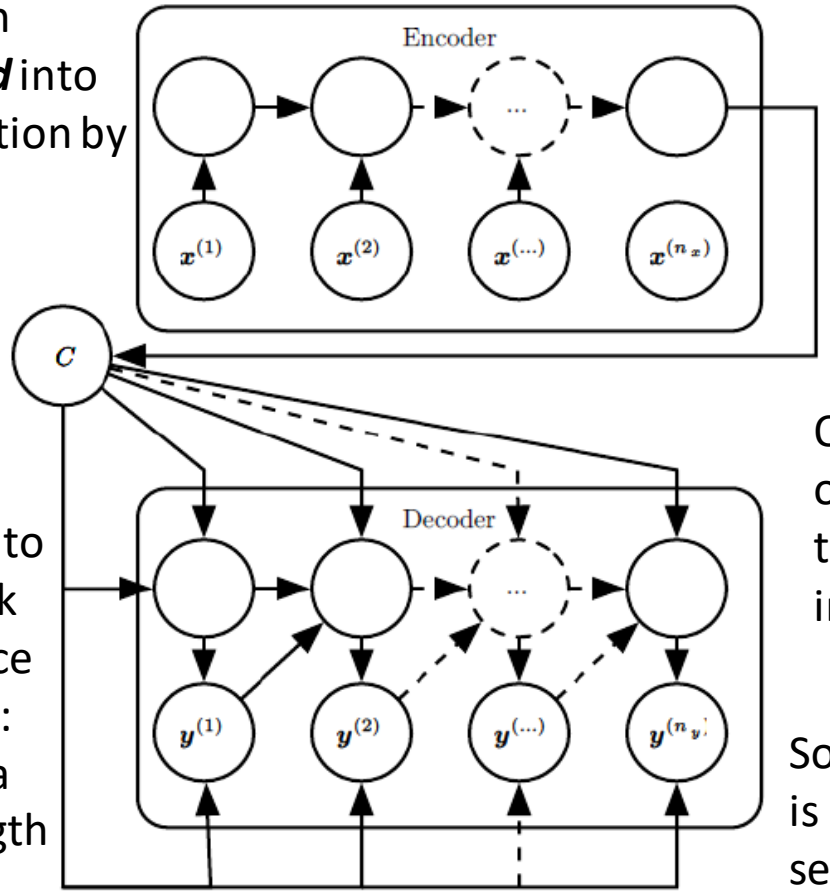
- Often, the input and target sequences may not be of the same length, nor even work on the same dictionary of symbols
  - Eg. machine translation
- Proposed Architecture
  - Encoder
  - Decoder
  - (Cho et al., 2014) *and* (Sutskever et al., 2014)

Encoder-Decoder

Inputs of arbitrary length (time-steps) are ***encoded*** into a distributed representation by a recurrent hidden layer

Distributed "context"

Context used as the input to another recurrent network which produces a sequence of output vectors (tokens): ***decodes*** the context into a sequence of arbitrary length (time-steps)



Often, we feed in the output sequence during training as an additional input (*teacher forcing*)

Sometimes, *teacher forcing* is not possible for certain sequence data, and the additional input can be omitted (eg. missing data)

Training Encoder-Decoder Networks

- For input sequences, create a one-hot dictionary for symbols (characters, words, etc.)
  - 0, 1 -> use [1, 0] and [0, 1]
- For output sequences, create a one-hot dictionary for symbols (doesn't have to be the same as the above
  - 0 ,1 start, stop -> use [1,0,0,0] [0,1,0,0] [0,0,1,0] and [0,0,0,1]
  - You need at least a stop token for most general cases
  - Indicates the end of the output sequence, and the decoder will learn to place this token at the end to indicate the end of a predicted sequence
  - Start token is also semantically useful for some problems (eg. learning to make two, three, or more consecutive sequences)
- During ***predictive decoding*** (not training), you can just prime the teacher forced input with the "start" token vector, and then *feed the predicted output back as the next input* (predicted teacher forcing signal) until reaching the "stop" token…

Random Embedding

- For discrete inputs, a useful trick to generate distributed encodings:
  - Random Vectors!
  - Torch.nn.Embedding

  https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html

- Holographic Reduced Representation
  - Plate, 1995
  - https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.6928
  - Can combine representations without growth in dimensionality via circular convolution

```
torch.nn.Embedding(
    num_embeddings: int,
    embedding_dim: int,
    padding_idx: Optional[int] = None,
    max_norm: Optional[float] = None,
    norm_type: float = 2.0,
    scale_grad_by_freq: bool = False,
    sparse: bool = False,
    _weight: Optional[torch.Tensor] = None,
    _freeze: bool = False,
    device=None,
    dtype=None,
) -> None

# Embeds integer inputs (0,1) into a 128-dimensional space
Torch.nn.Embedding(2, 128)
```