Recurrent and Convolutional Neural Networks

CSCI-P556 Applied Machine Learning
Lecture 17

D.S. Williamson

Agenda and Learning Outcomes

Today's Topics

- **Topics**:
  - Finish discussion on regularization
  - Recurrent Neural Networks (RNNs)
    - Understand how temporal dependencies are incorporated into deep neural networks
    - Explain Backpropagation through time (BPTT)
  - Convolutional Neural Networks (CNNs) (time permitting)

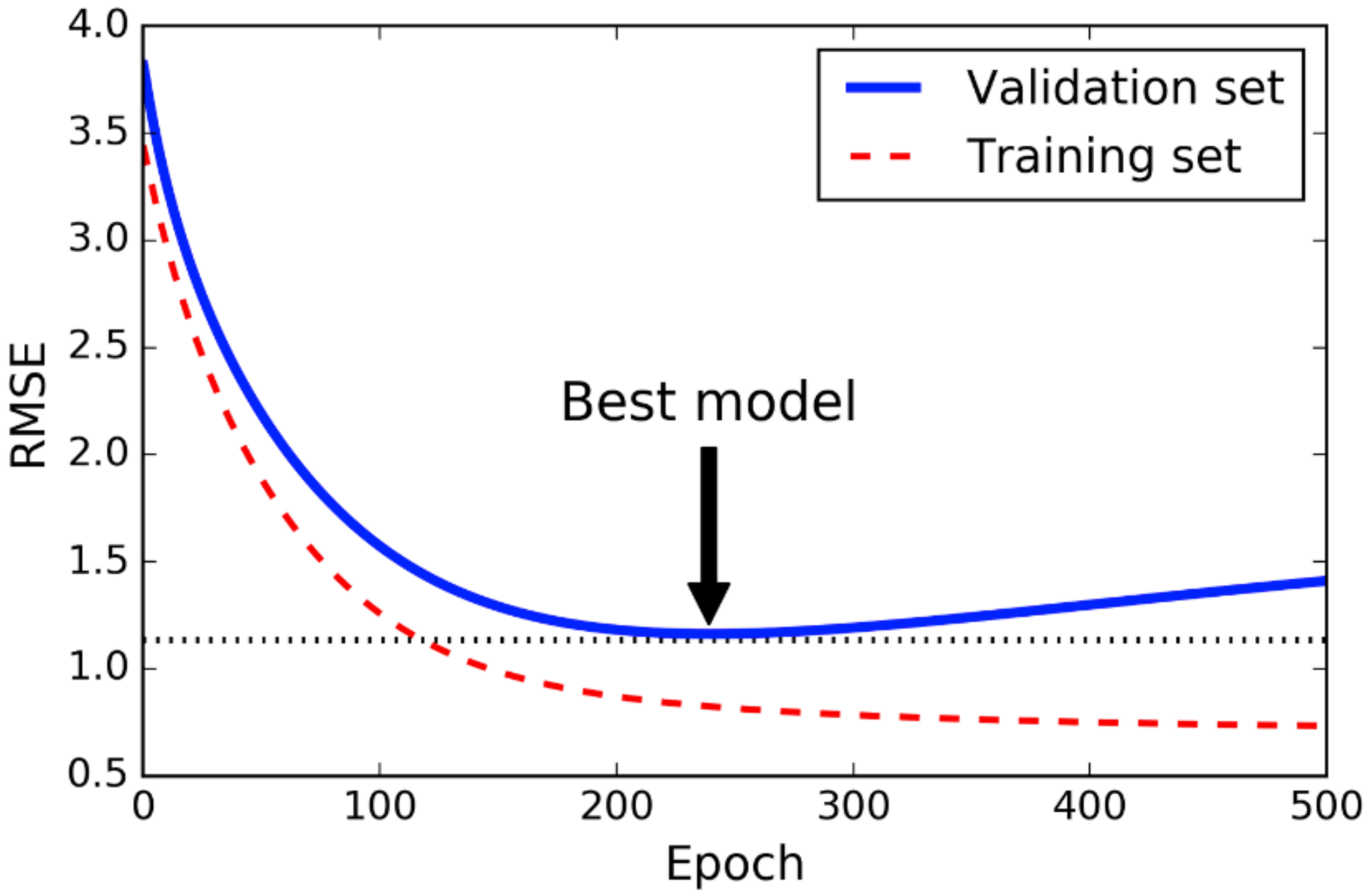
- **Announcements**
  - Quiz #2 next week on Thursday

Regularization: For Improving Generalization

Early Stopping

An approach to improve generalization

- **Idea**: avoid overfitting, by stopping training when performance on the validation set starts getting worse
- **Basic steps**:
  - Evaluate model on validation set every N epochs
  - Save the model if performance is “better” than before
  - Count the number of steps since the last model was saved, and stop training when this number reaches a pre-defined limit (e.g. 3*N)
- Alternatively, keep track of the “best” performing model and run training for all epochs. Use the “best” performing model during testing. This is known as ***model selection***



L_1 and L_2 Regularization

Constrain solution space (weight values) during training to avoid overfitting

- **Idea**: Add a regularization term, $R(\theta)$, to the cost function (e.g. MSE) that is scaled by a factor, α, that controls how much you want to regularize

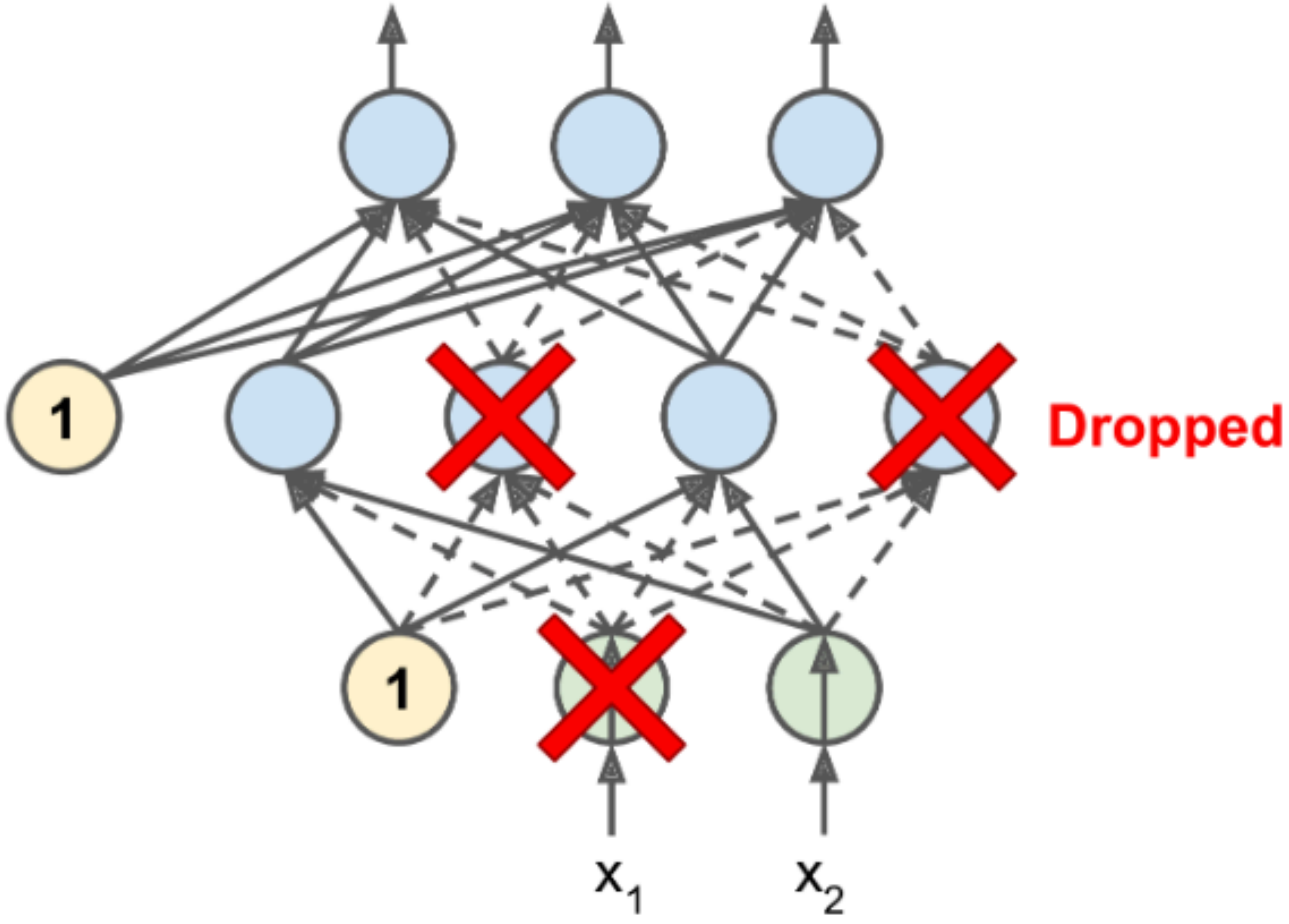
$$\text{Cost}(\theta) = MSE(\theta) + \alpha R(\theta)$$

  - **Ridge Regression** (L_2 Regularization): Goal is to keep model weights as small as possible. $R(\theta) = ||\theta||_2^2$
  - **Lasso Regression** (L_1 Regularization): Goal is to zero the weights of the least important features. $R(\theta) = \sum_{i=1}^{n} |\theta_i|$
  - **Elastic Net**: Mix of Ridge and Lasso regularization terms. $R(\theta) = rR_{ridge}(\theta) + \frac{1-r}{2}R_{LASSO}(\theta)$. Control mix with *r* (r = 0 means LASSO, r = 1 means Ridge).

Dropout

Avoid over-reliance on certain neurons. Improve adaptability

- Randomly ignore a subset of neurons during training (excludes output layer)
- Every neuron has a probability of being dropped (outputs ignored) **during each training step.**
    - Controlled by drop out rate
    - Perform “coin flip” for each neuron to determine if it will be dropped
- **Downside**: May slow down convergence




```
torch.nn.Dropout(p=0.5, inplace=False)
```

Other Training Considerations

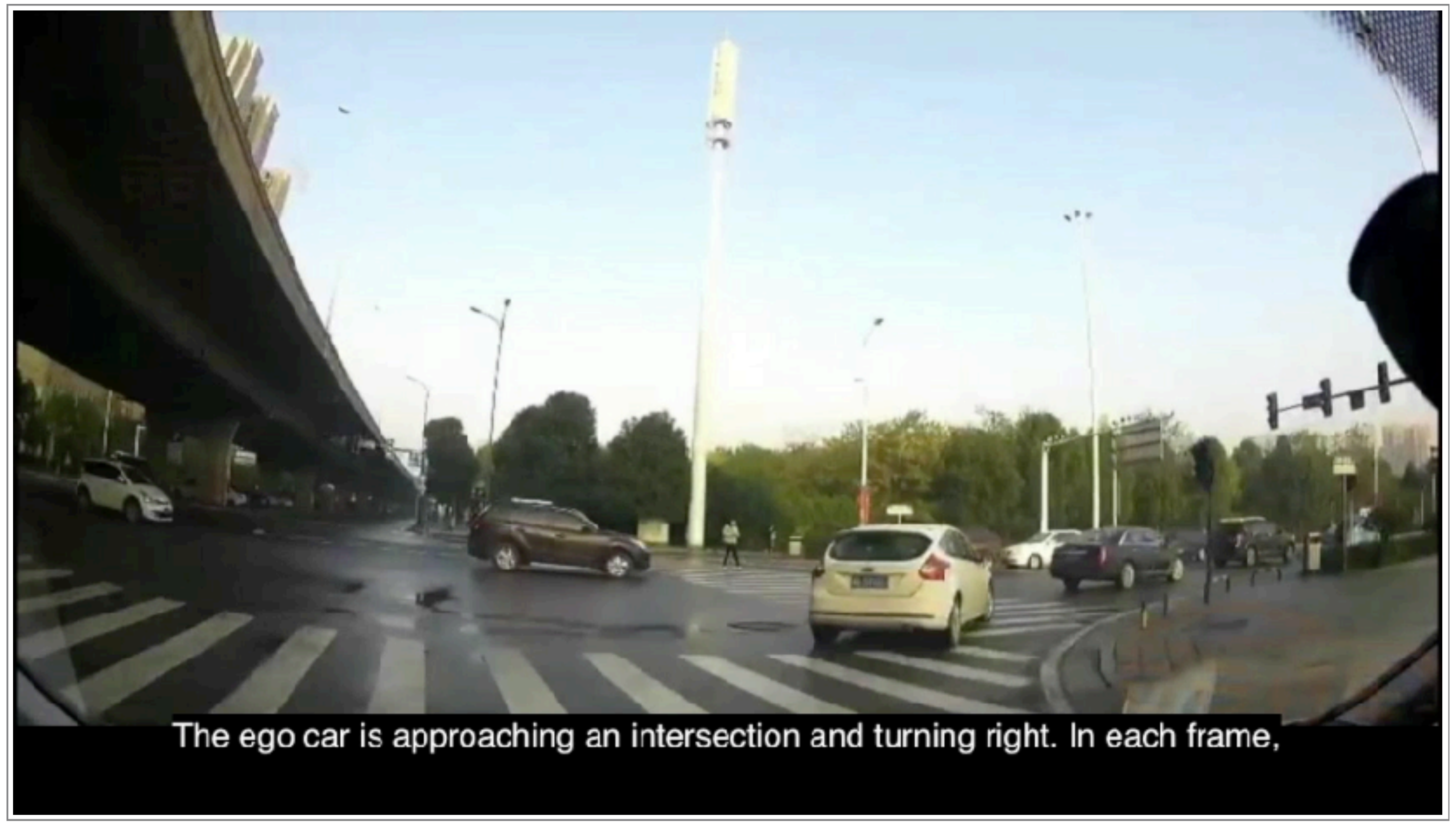
- **Reusing pre-trained network layers** (e.g. transfer learning) - speeds up training and requires less data
- **Freezing lower-layers of already-trained models** — may have detected low-level features for the problem
- **Unsupervised layer-by-layer Pre-training** — useful when don't have a lot of labeled data. Often uses auto encoders, but previously used Restricted Boltzmann Machines (RBM)
- …

DNNs: Considering Time

Example Sequential Data

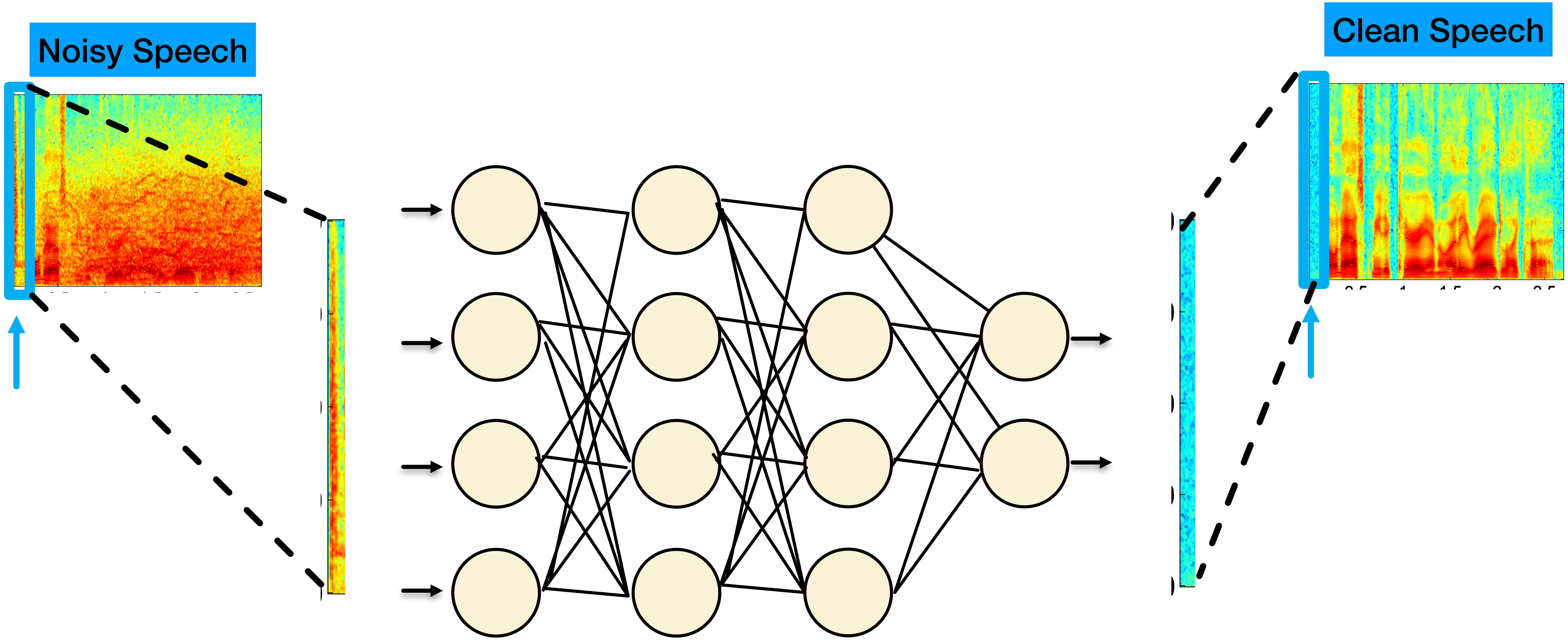
Time Varying Data

- So far we haven't formally considered data that varies over time (e.g. video, audio, health monitored data,…) or is sequential in nature (e.g. text)
- This data will have features and labels that evolve over time (or the sequence), many of which may be correlated
  - For example,
    - Action Recognition in Videos ->
    - Live Transcription from Audio



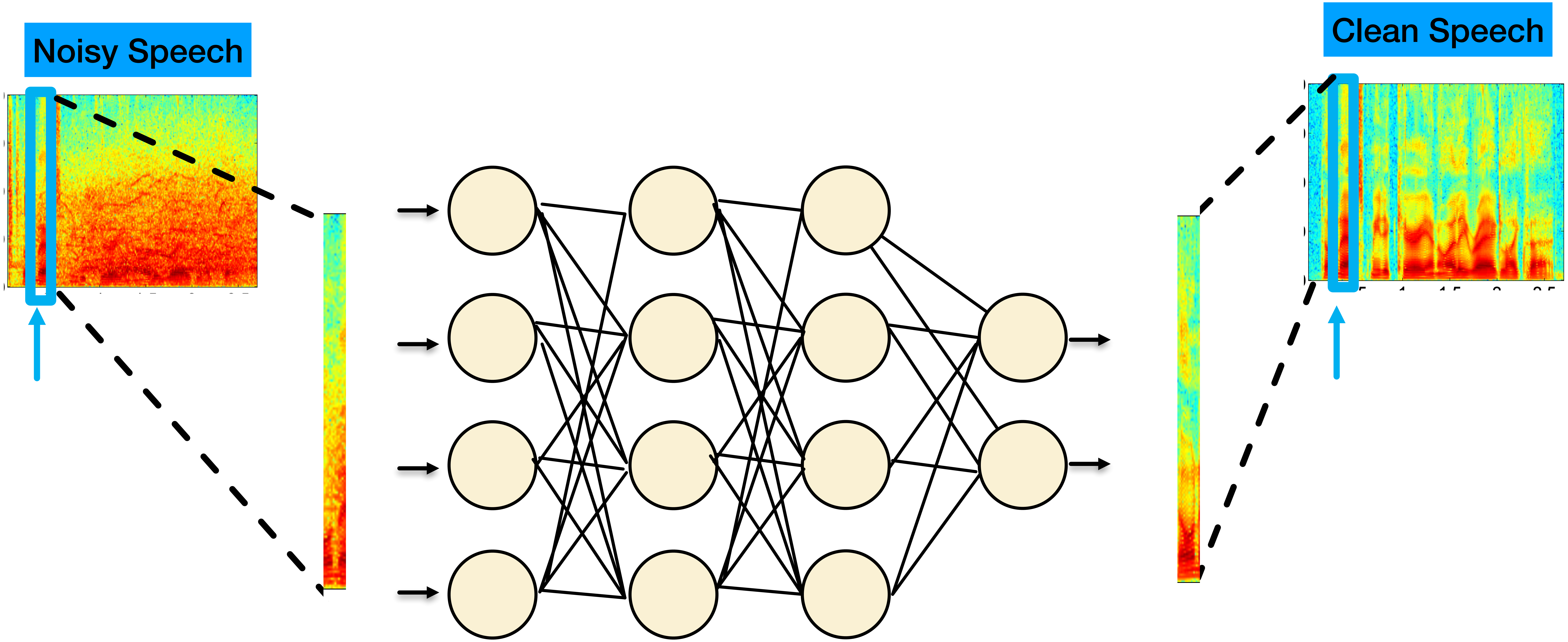
DNN for Speech Enhancement

- You work for a hearing-aid design company, and you want to use a DNN to remove unwanted background noise. This is generally down independently for small segments of speech



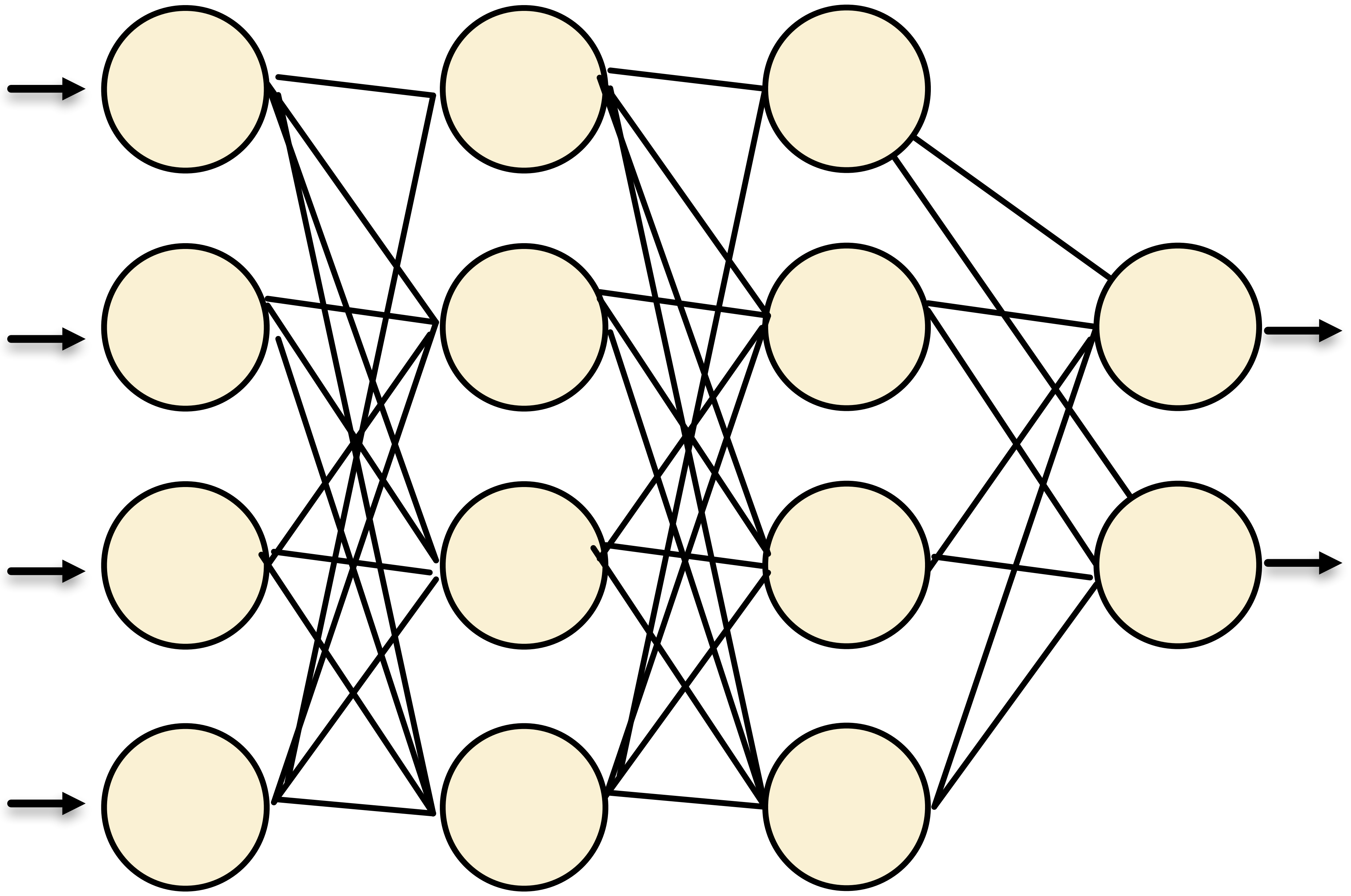
DNN for Speech Enhancement

- You work for a hearing-aid design company, and you want to use a DNN to remove unwanted background noise. This is generally down independently for small segments of speech



Erroneous Assumptions ?

- What assumptions about the input and output are these models making?
  - **Single input dependency**: Network state depends on current input
  - **Single output dependency**: Neurons current state (output) is independent of its previous
  - **Layer-by-layer dependency**
    - Current layer depends only on the previous layers outputs, at the current sample



Addressing DNN Limitations

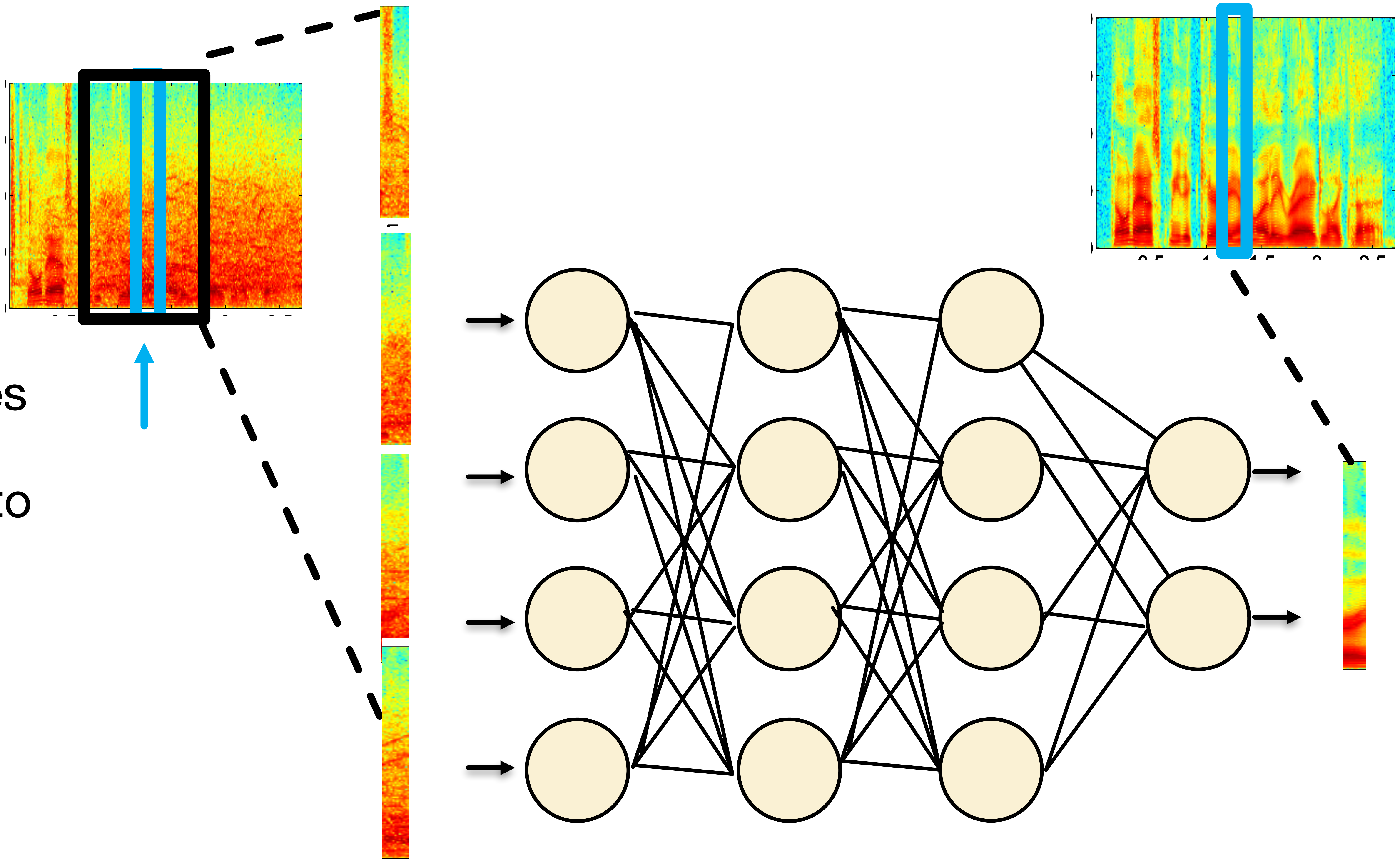
- Layer-by-layer dependencies can be addressed with recurrent neural networks (mainly LSTMs) and/or skip connections (more on the first later)

- **How can we address the first two assumptions? (single input and single output)**

Data Splicing

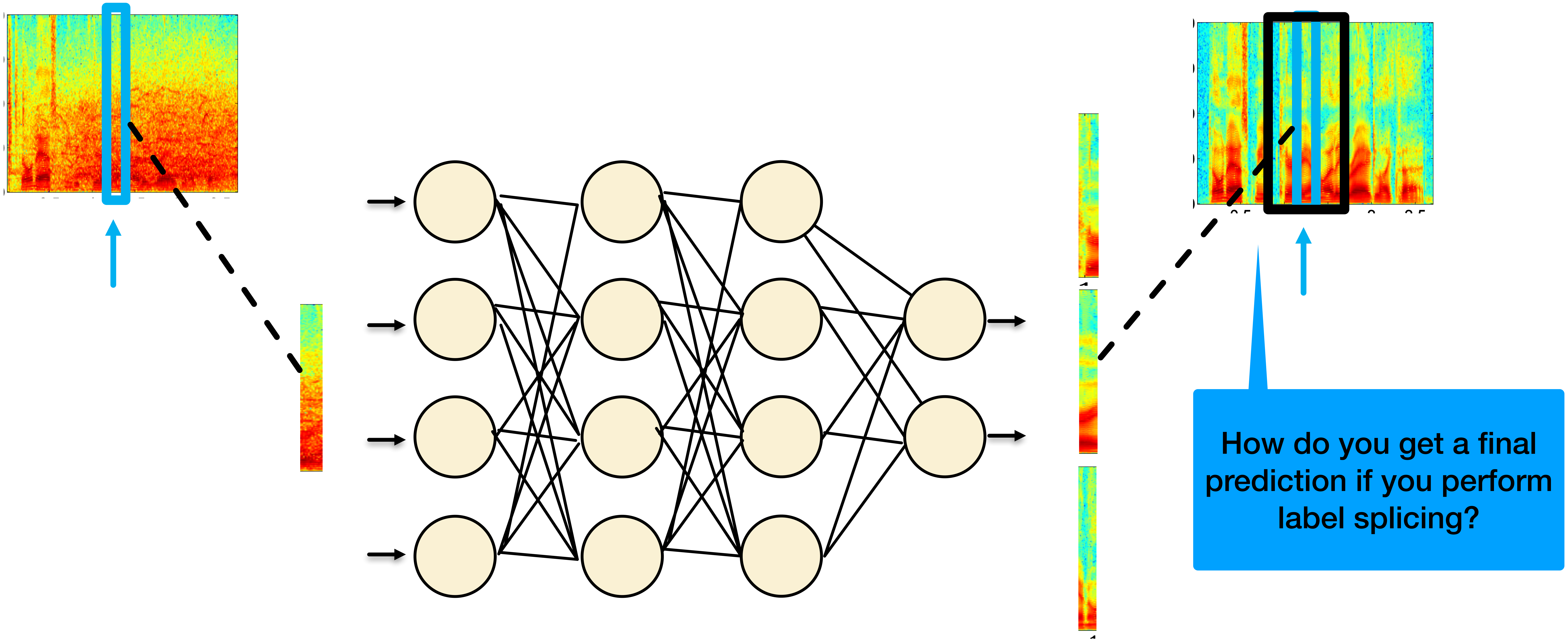
For Features

- Concatenate multiple time frames into a single data vector (feature or label)
- ***<u>Input feature splicing</u>***
    - Include adjacent feature frames
    - No limit on number of frames to include



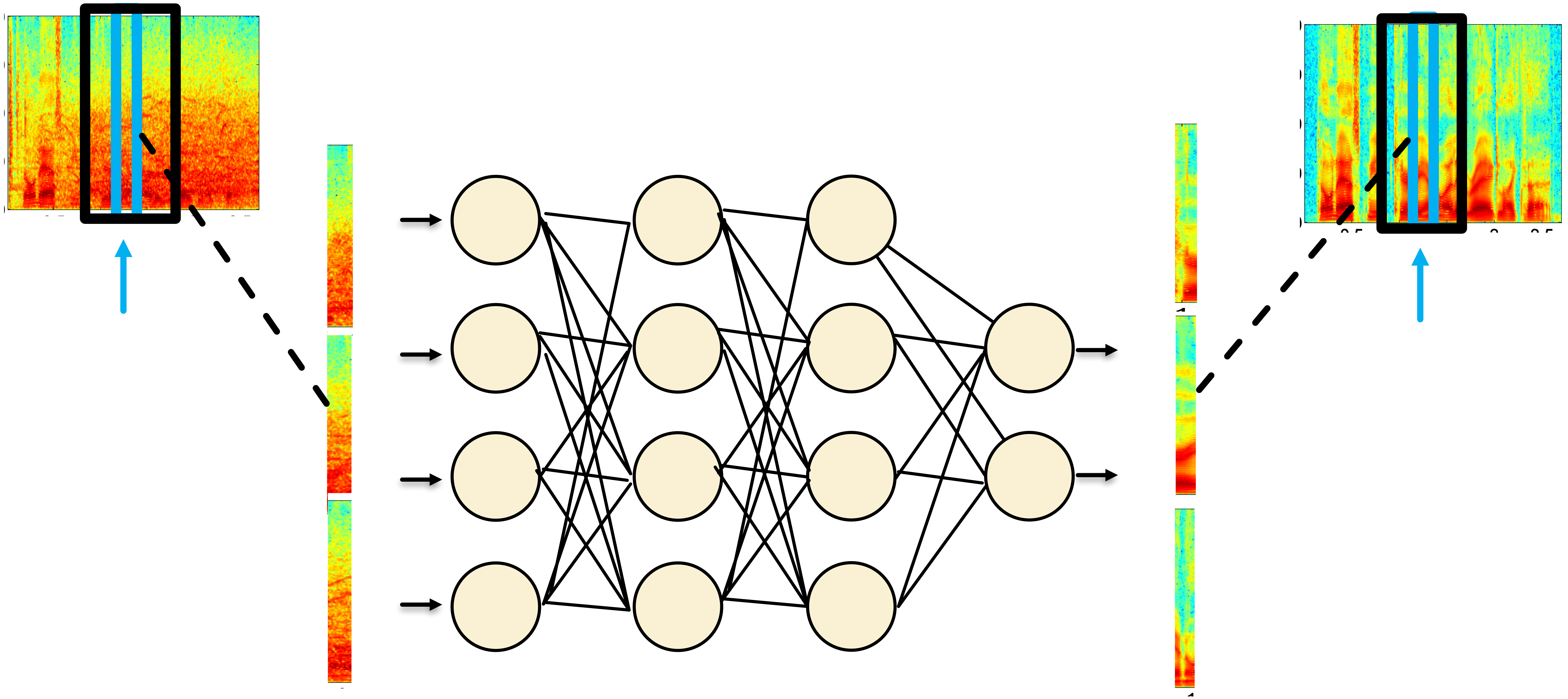
Data Splicing for Speech

For Labels



Data Splicing for Speech

For Features and Labels



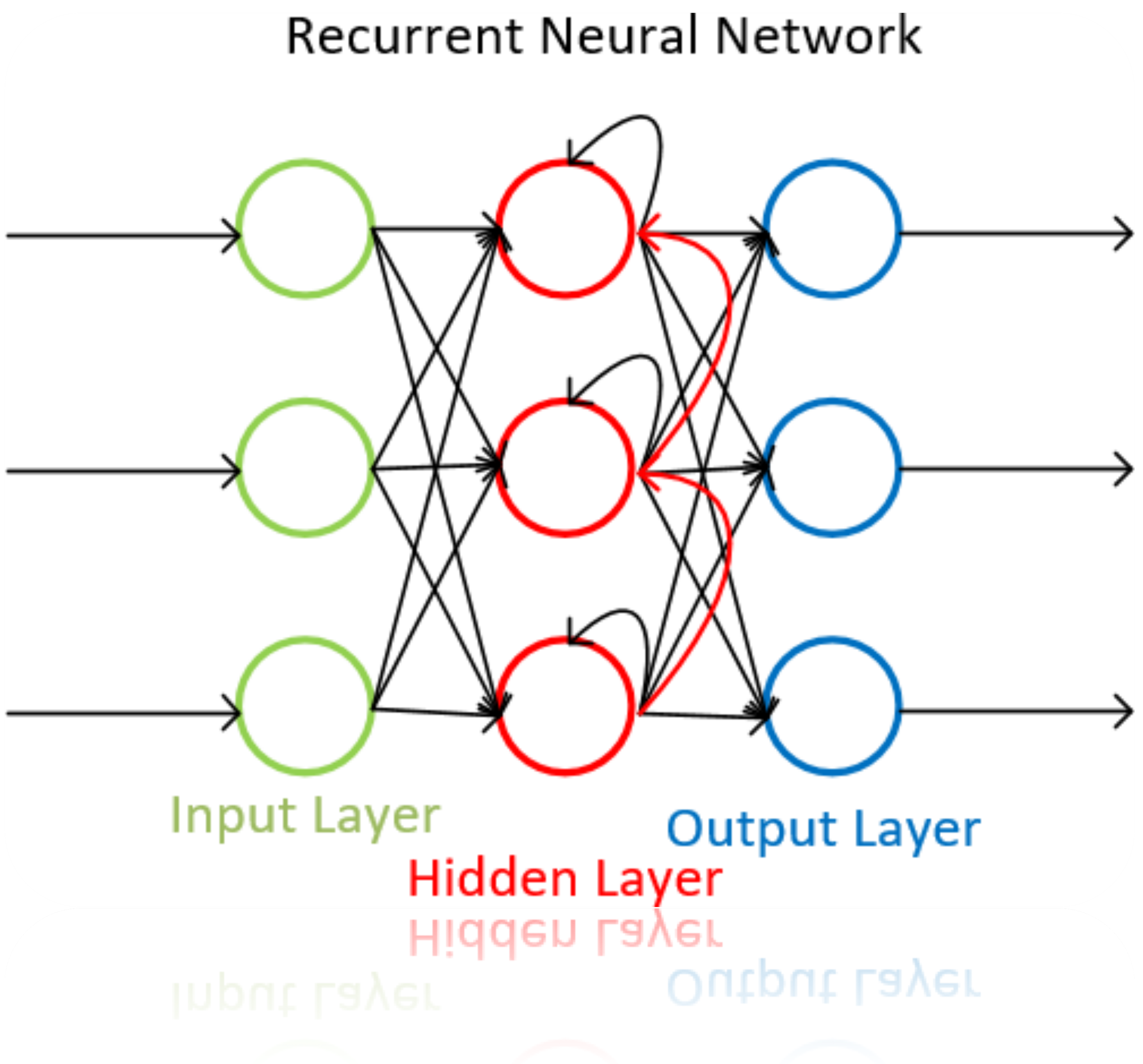
Things to Consider

Data Splicing

- **How many frames should be included in the features/labels**?
    - This impacts network architecture (e.g., more units in output layer; may need more in hidden layers)
    - Another hyper-parameter to tune
- **Issues with Data Splicing**
    - Less efficient computationally and uses more resources (e.g. memory)
    - Increased Latency. Is “real-time” processing desired?
    - Network does not “enforce” or “learn from” these correlations

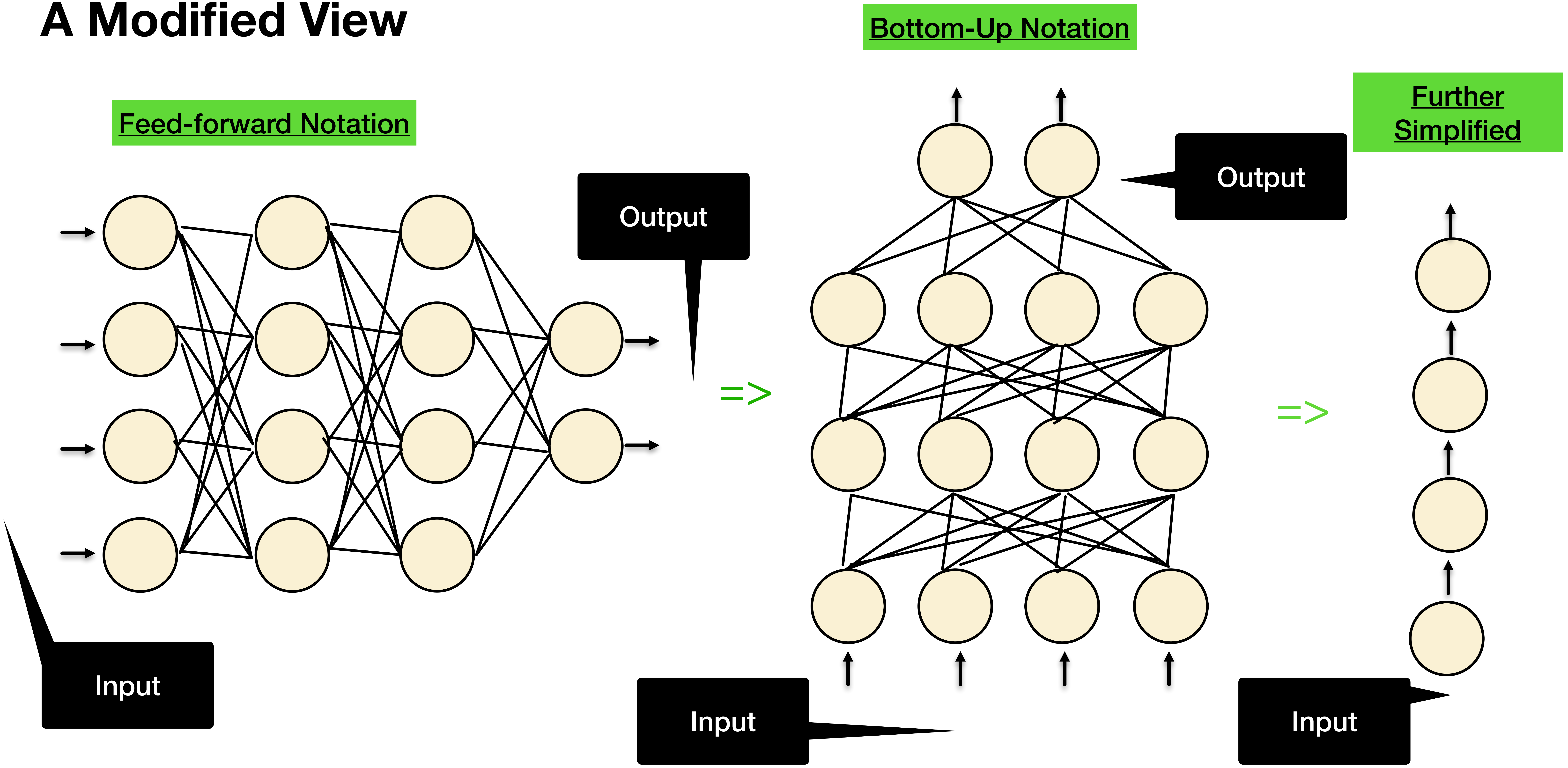
Recurrent Neural Networks (RNNs)

- RNNs have “memory”
  - Remember prior calculations
  - This is done through loops (or cycles)
  - Current output value is seen one-time step (or sequence) later
- RNNs are ideal for handling sequential data
  - Traditional neural networks assume that all inputs (and outputs) are independent of each other
  - This is not true for many tasks (i.e. speech recognition, natural language processing)



Network Notation

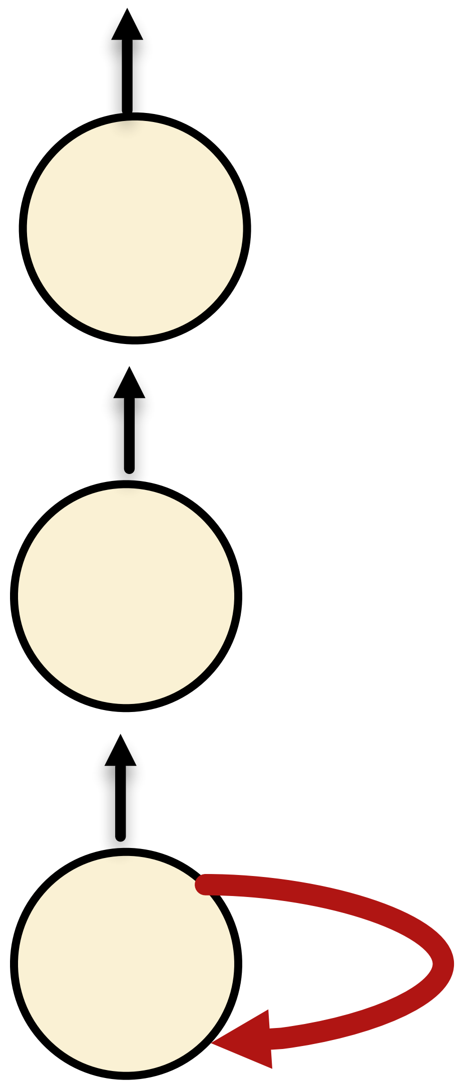
A Modified View



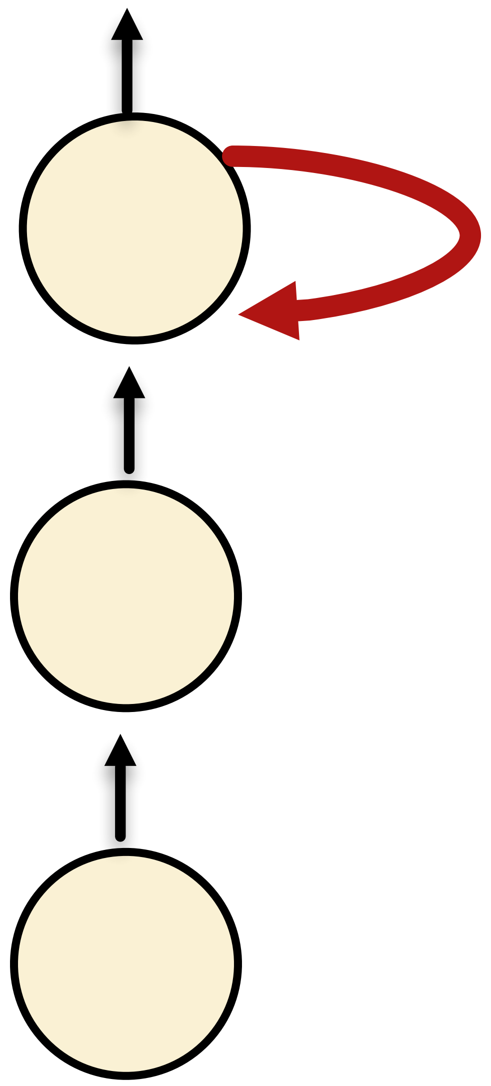
Enforcing Correlations in the Network

- **Temporal (or sequential) Correlations** can be enforced by adding recurrent connections within the network

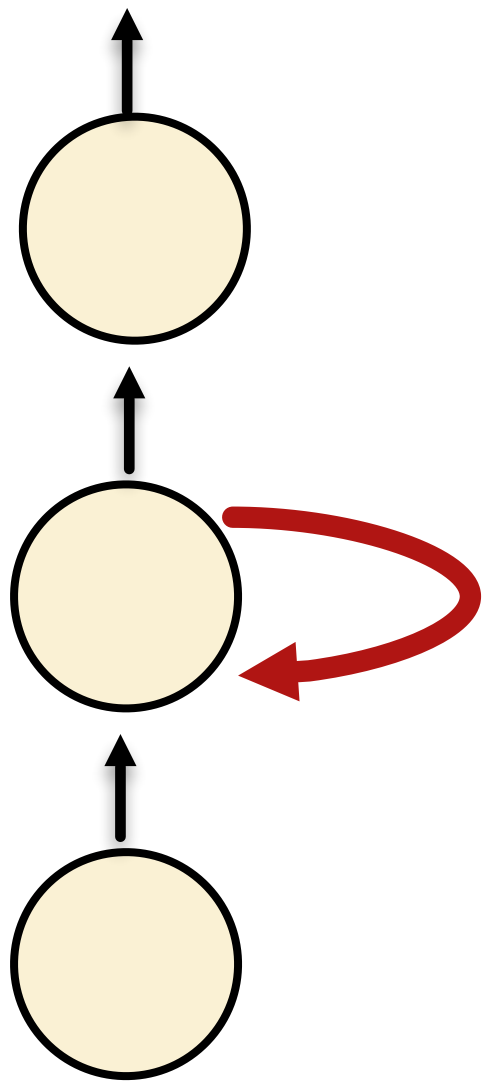
Ex. Input layer Recurrency



Ex. Output layer Recurrency



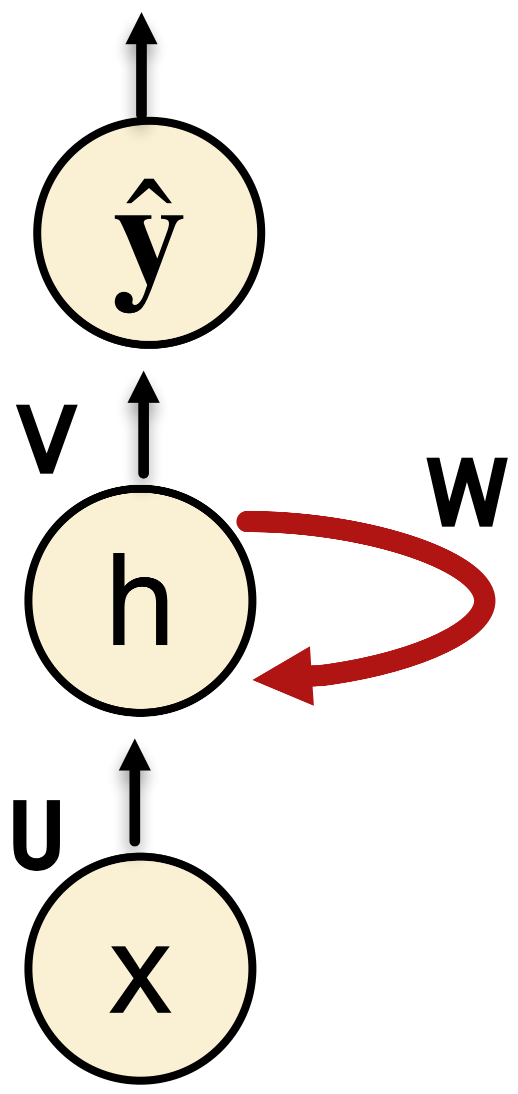
Ex. Hidden layer Recurrency



Recurrent Hidden Units (RHU)

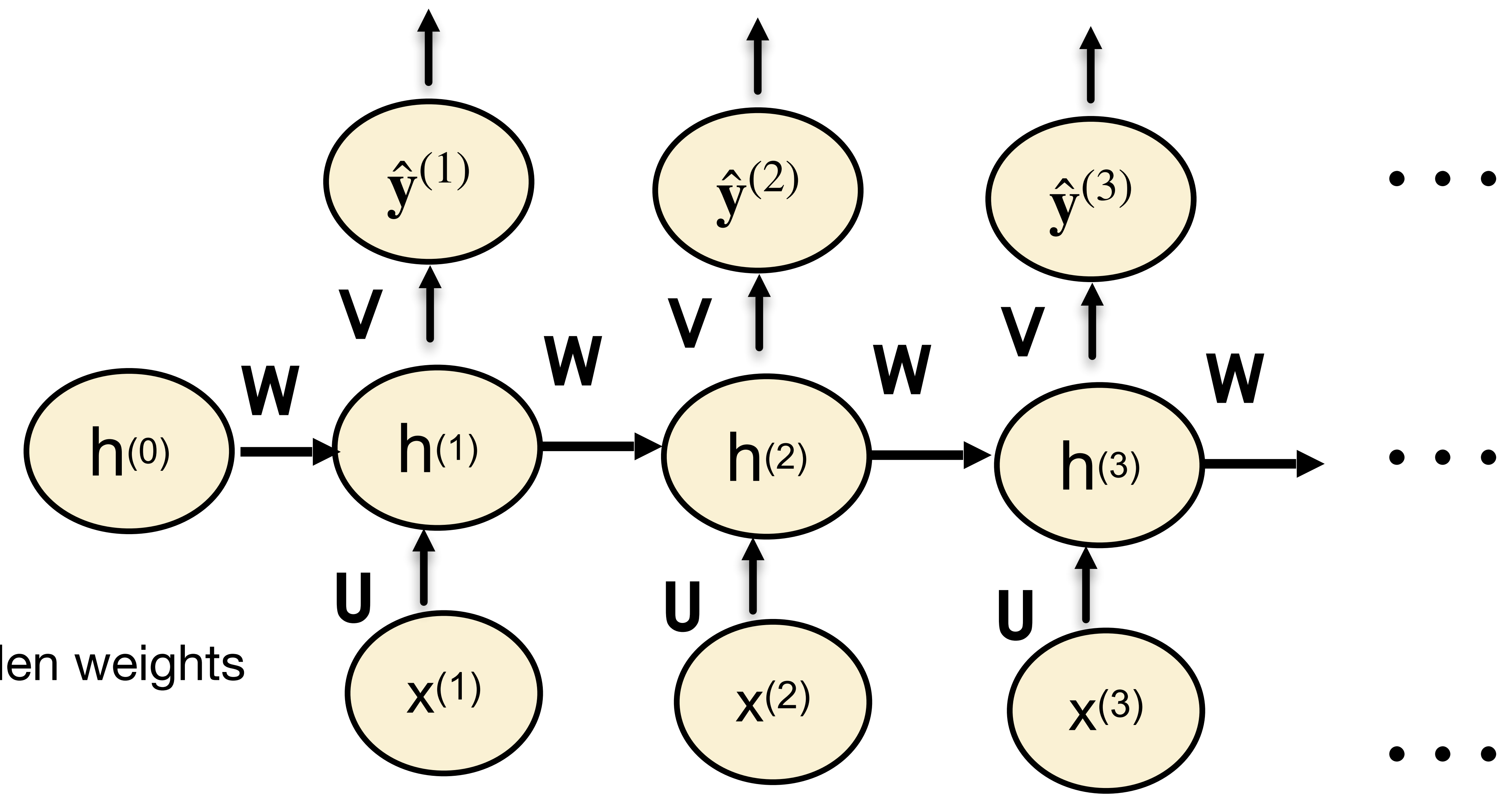
Forward pass of RNN with RHU: Unfold over time

- **Example of *sequence-to-sequence* mapping**
- Assume a sequence of length T (e.g. there are T separate input features and T separate corresponding labels - e.g., $x^{(t)}$ represents a frame of a video)



Unfolding in time

=>



- **U**: input weights
- **W**: hidden-to-hidden weights
- **V**: output weights
- **$\hat{\mathbf{y}}$:** network output
- **h:** hidden layer output
- **x**: network input

$$\mathbf{v}^{(t)} = \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)}$$

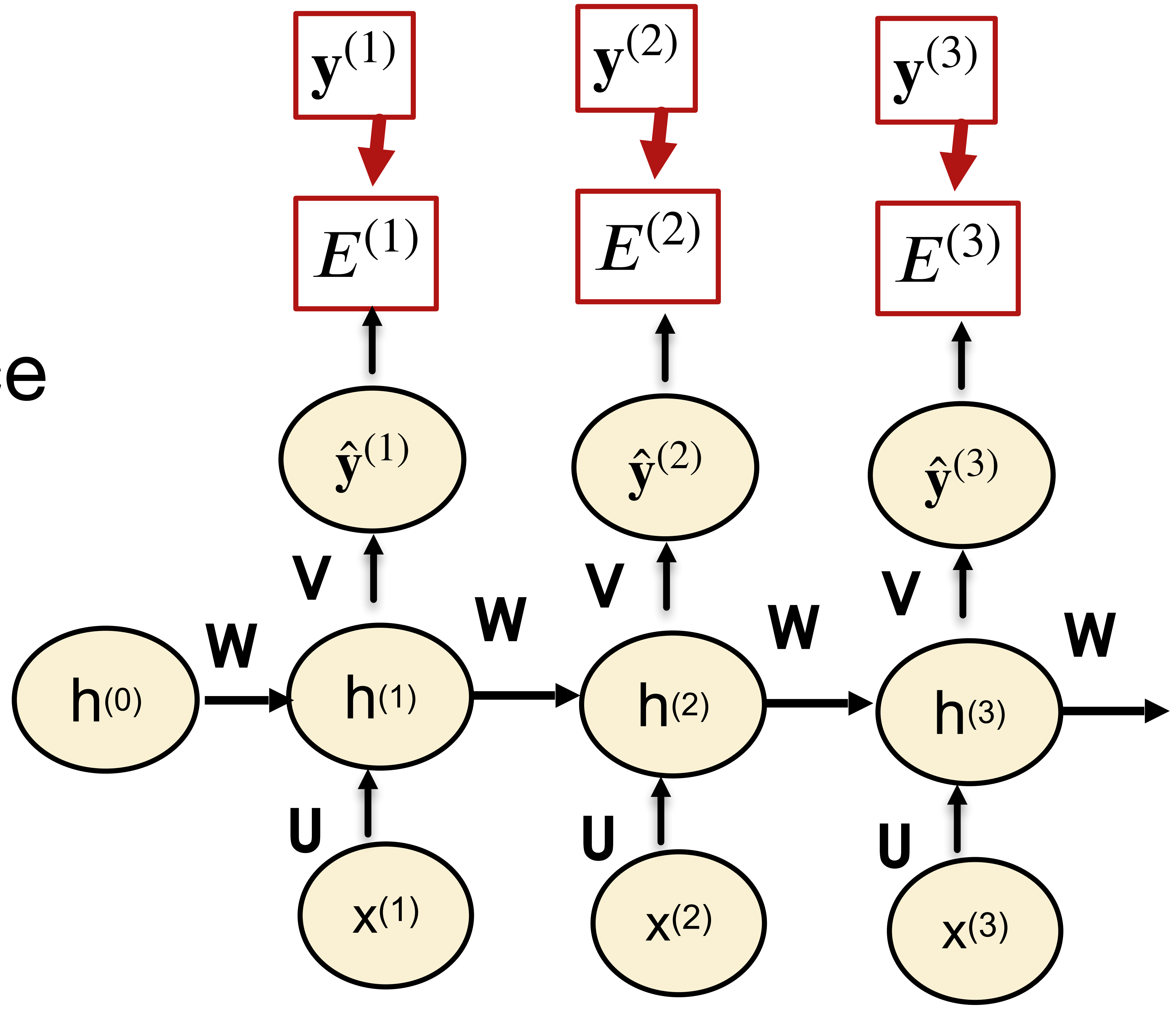
$$h^{(t)} = \phi(\mathbf{v}^{(t)})$$

$$\hat{\mathbf{y}}^{(t)} = \phi(\mathbf{V}\mathbf{h}^{(t)} + c)$$

b and **c** are bias terms
$\mathbf{h}^{(0)}$ is initialized to 0

Image adopted from Goodfellow, 2016

RHU Error Calculation



- Total error is calculated across all sequence outputs (e.g. all videos, not just one)
- Assume MSE loss function, with $\mathbf{y}^{(t)}$ the desired output at time ***t***
- ***Element-wise error*** *(loss function)*

$$E^{(t)}(\hat{y}^{(t)}, y^{(t)}) = \frac{1}{2}(y^{(t)} - \hat{y}^{(t)})^2$$

- ***Sequence Error (e.g. individual video)***

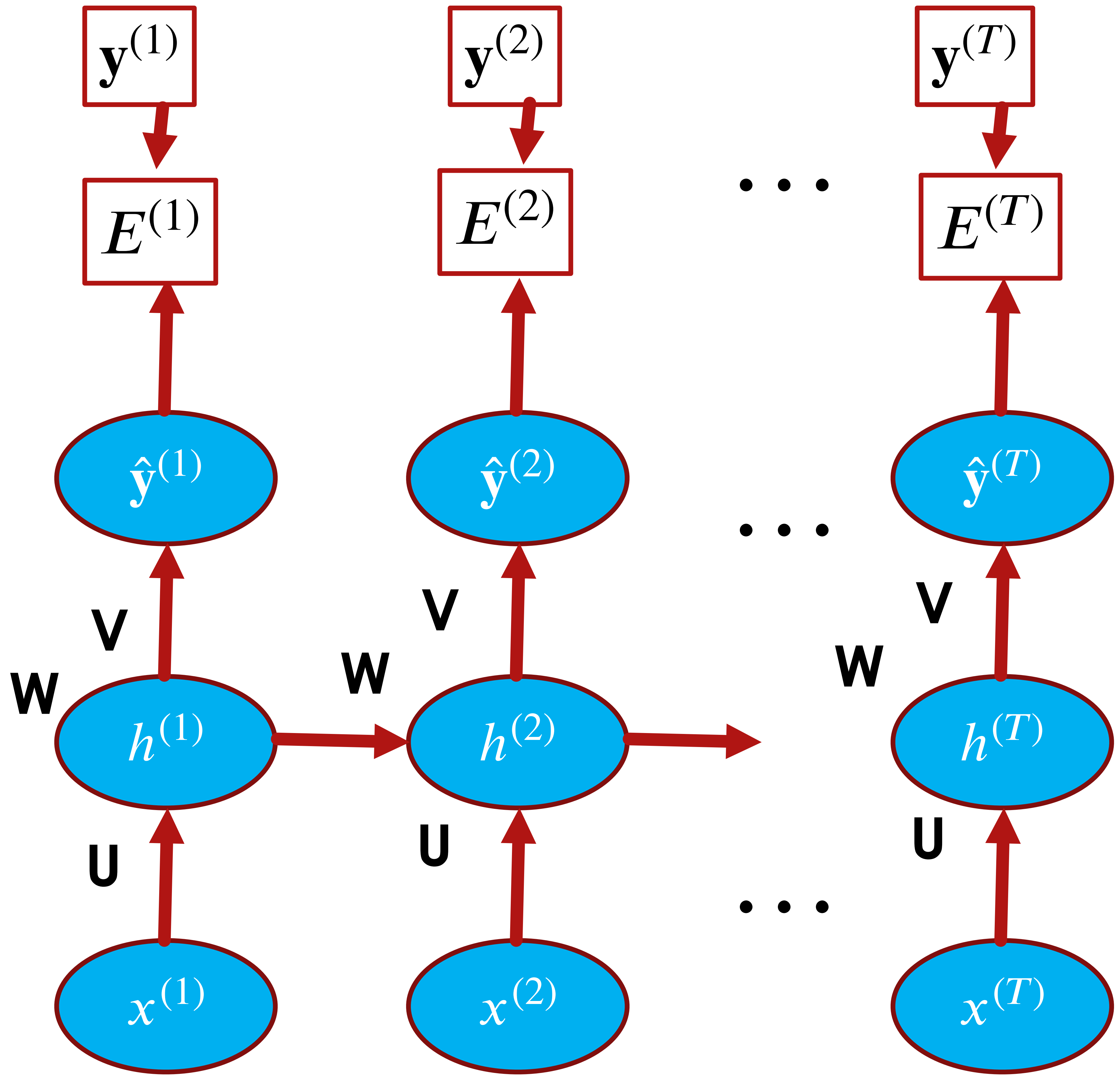
$$E_s = \sum_{t=1}^{T} E^{(t)}(\hat{y}^{(t)}, y^{(t)})$$

- ***Total Data Error (across all sequences)***

$$E_{total} = \sum_{s=1}^{S} E_s$$

Backpropagation through Time (BPTT)

- Treat the unfolded RNN as a DNN (e.g. feed-forward network with multiple hidden layers, and multiple output layers)
- Weights, however, are the same for each "layer", unlike a DNN
- Propagate the error backwards as before, but now it is through time



Weight Update with RHU

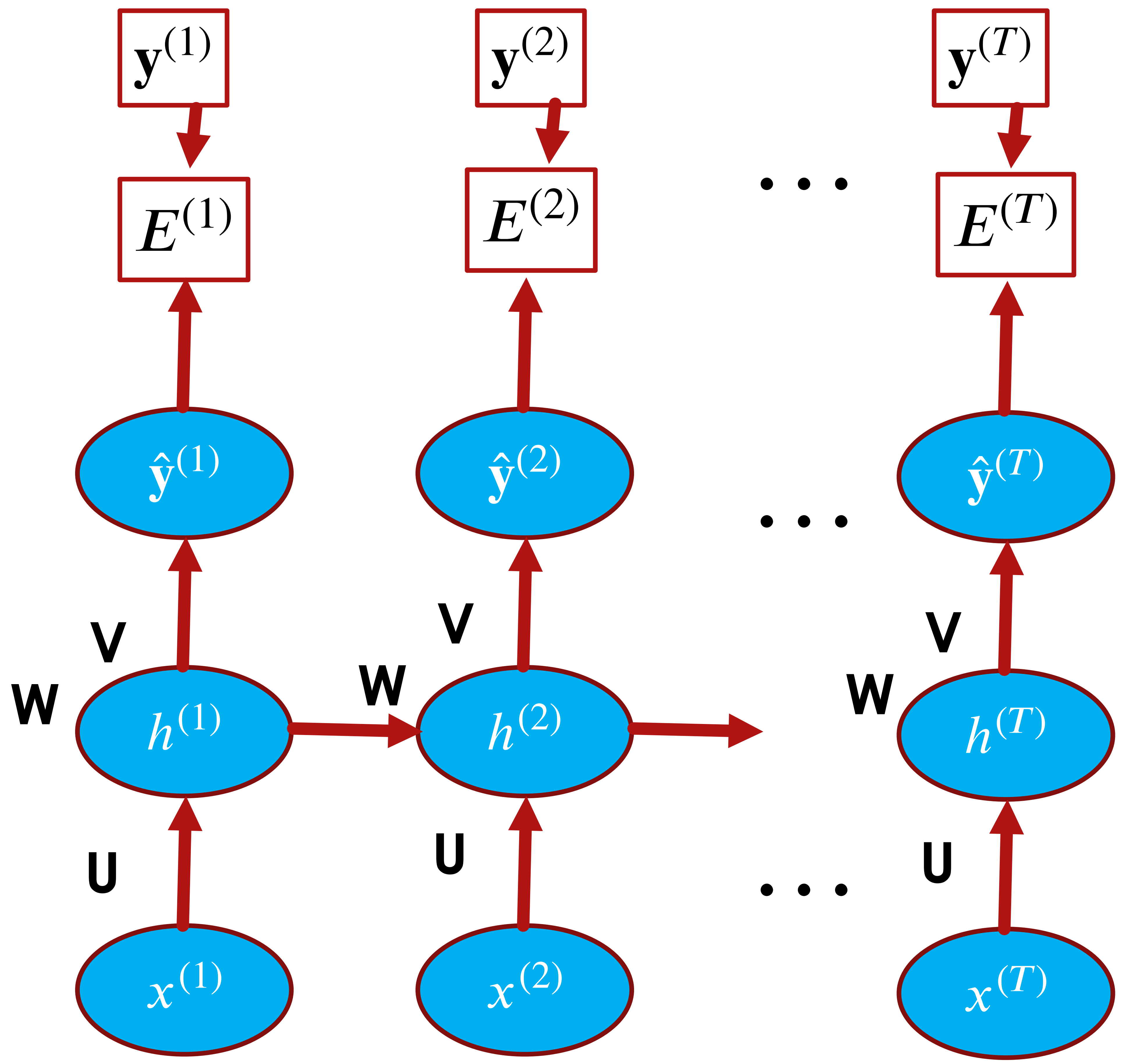
- For a single-hidden layer RNN
  - Output layer

$$V(n+1) = V(n) + \eta_V \sum_{t=1}^{T} \delta_{\hat{\mathbf{y}}}^{(t)} \mathbf{h}^{(t)}$$

  - Hidden Layer

$$W(n+1) = W(n) + \eta_W \sum_{t=1}^{T} \delta_h^{(t)} \mathbf{h}^{(t-1)}$$

  - Input Layer: $U(n+1) = U(n) + \eta_U \sum_{t=1}^{T} \delta_h^{(t)} \mathbf{x}^{(t)}$

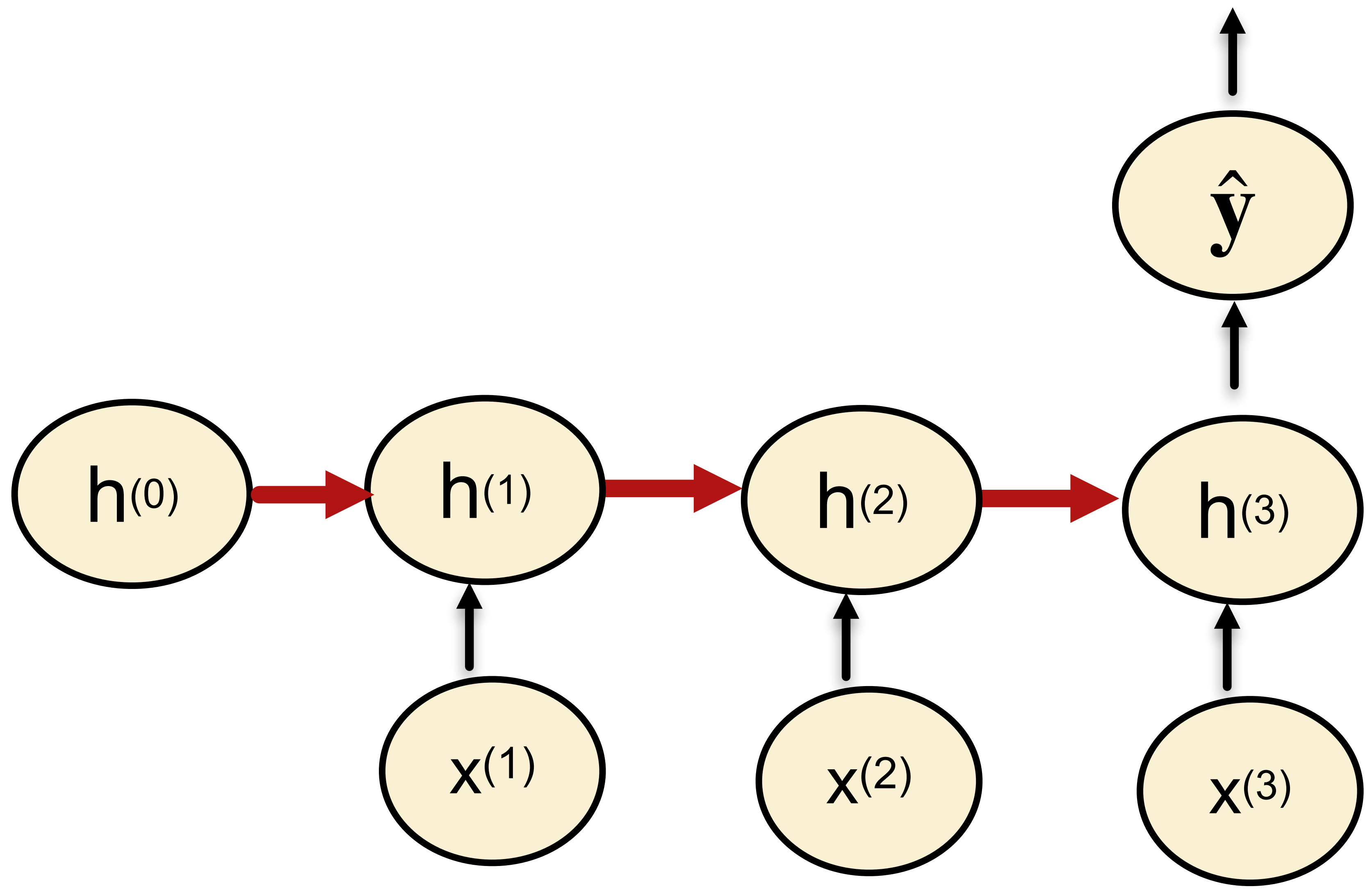


Sequential Input, Single Output

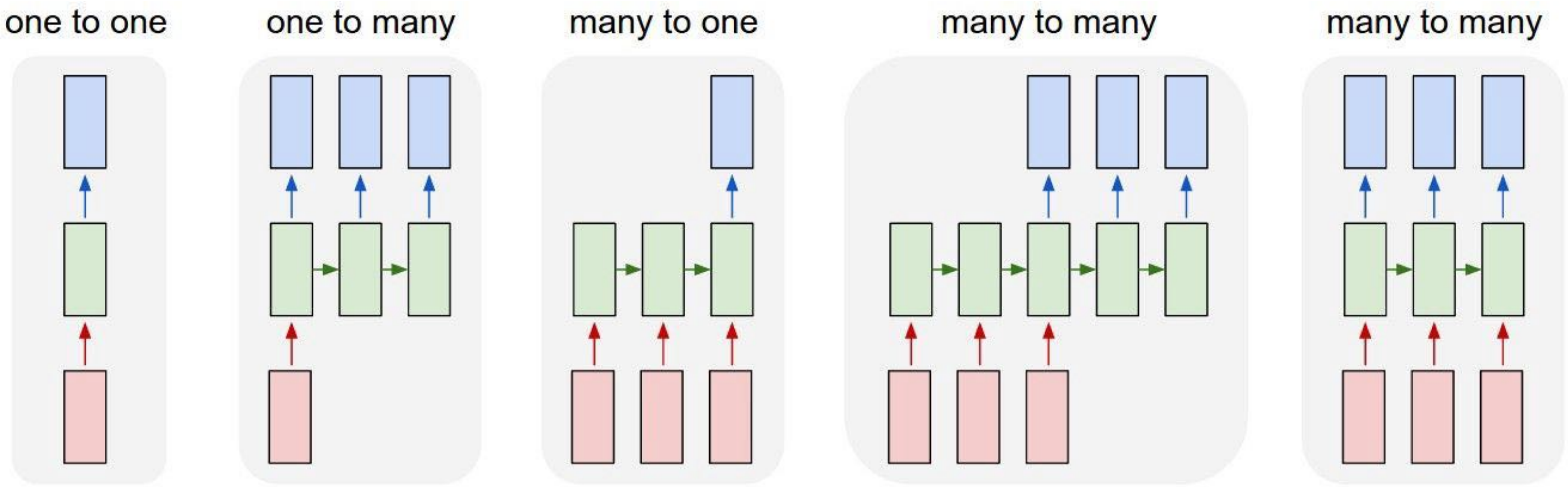
- Time unfolded RNN with a single output at the end of the sequence.

- This network is useful for summarizing a sequence and producing a fixed-size representation, which may be useful for further processing

- What application would this be useful for?



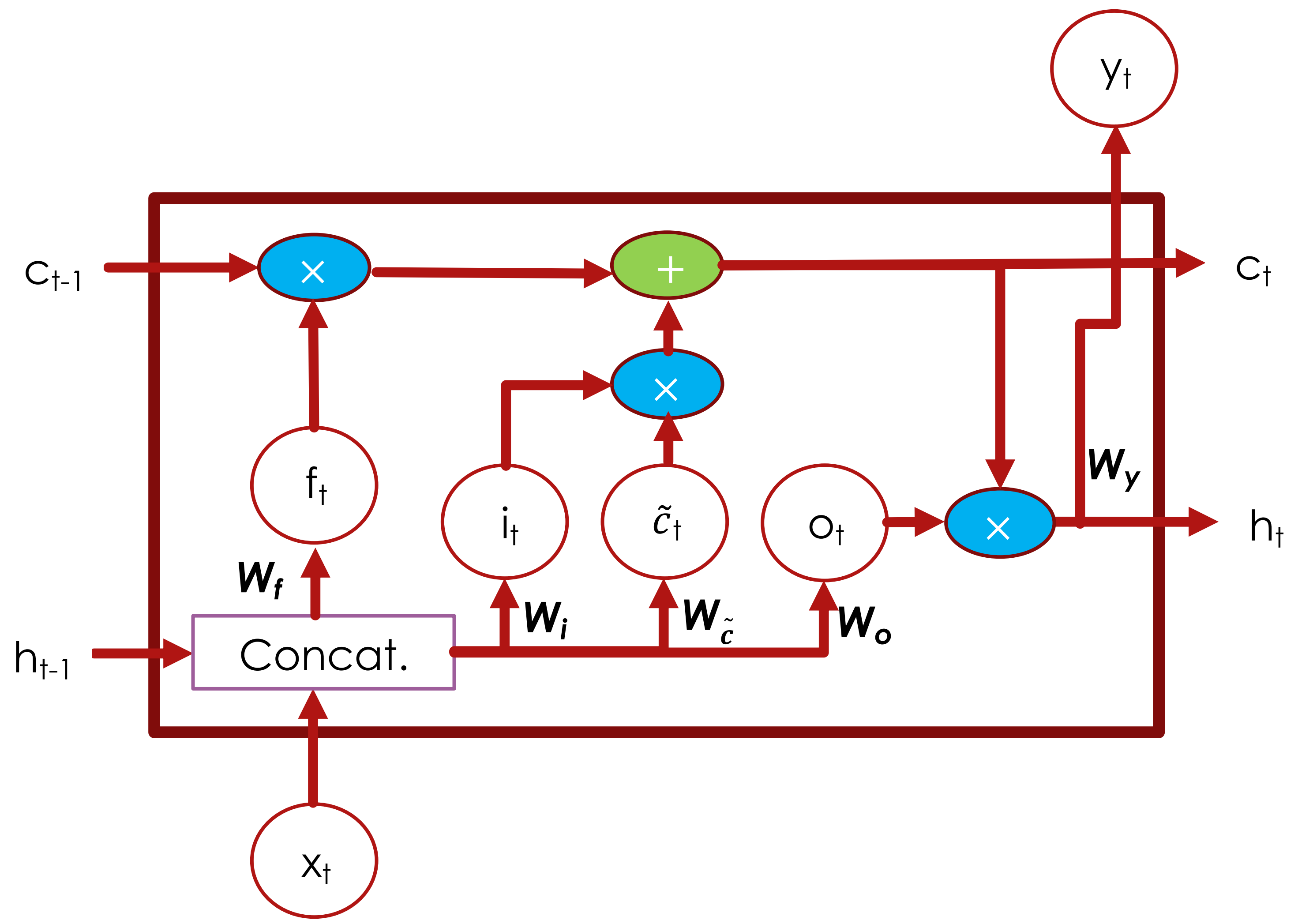
Other RNN Configurations



Limitations of RNNs

- Training RNNs is difficult
  - ***Vanishing gradient***: gradient contributions from well before T (sequence length) become 0
    - States at earlier steps do not contribute and are not updated
  - ***Exploding gradient***: gradient becomes to large
    - Causes program to crash
    - Must clip gradients

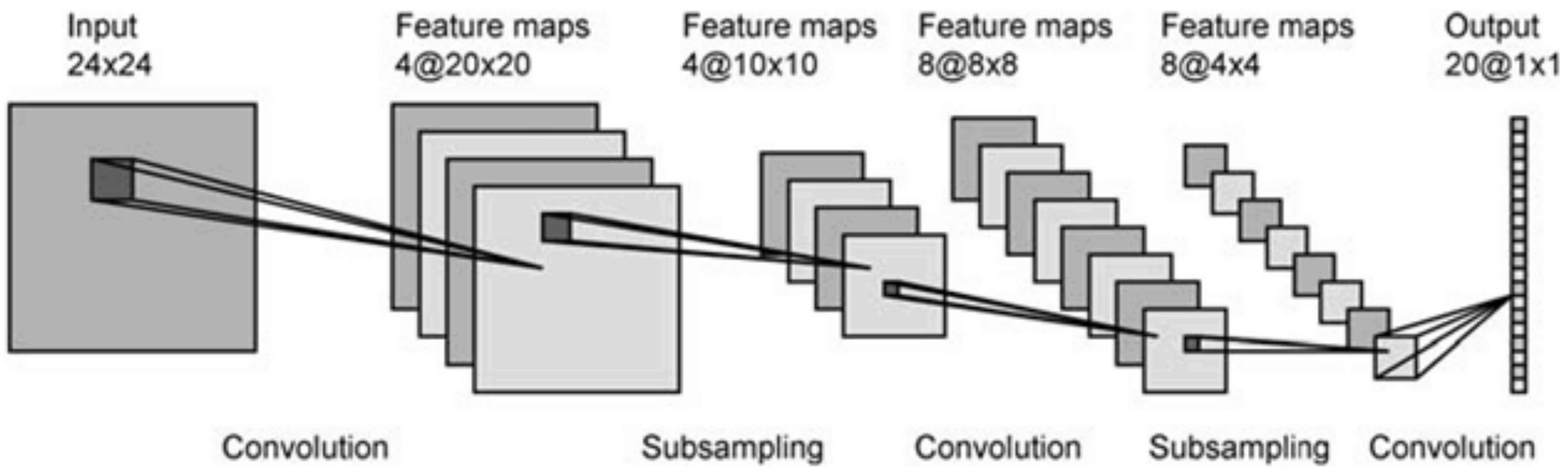
- RNNs do not capture long-term dependencies
  - They have sort memories. Current example captures a single time delay
  - Several data applications, including speech, have long-term dependencies
    - Interactions of words that are several steps apart
    - Long-short term memory (LSTM) RNNs are used instead



Convolutional Neural Networks

Convolutional Neural Networks (CNNs)

- Used for processing data with grid-like topology (i.e. images). Networks use convolution in place of general matrix multiplication
- There are four main operations in CNNs
  - Convolution
  - Nonlinear Activation Function (i.e. ReLU)
  - Pooling (or Subsampling)
  - Classification



Convolution

- Convolution is a linear mathematical operation on two functions

- Given functions $x(t)$ and $w(t)$, the convolution of $x(t)$ and $w(t)$ is as follows

$$s(t) = x(t) * w(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

- This step is used to extract "features" from an input, so it is also referred to as the ***feature map*** stage

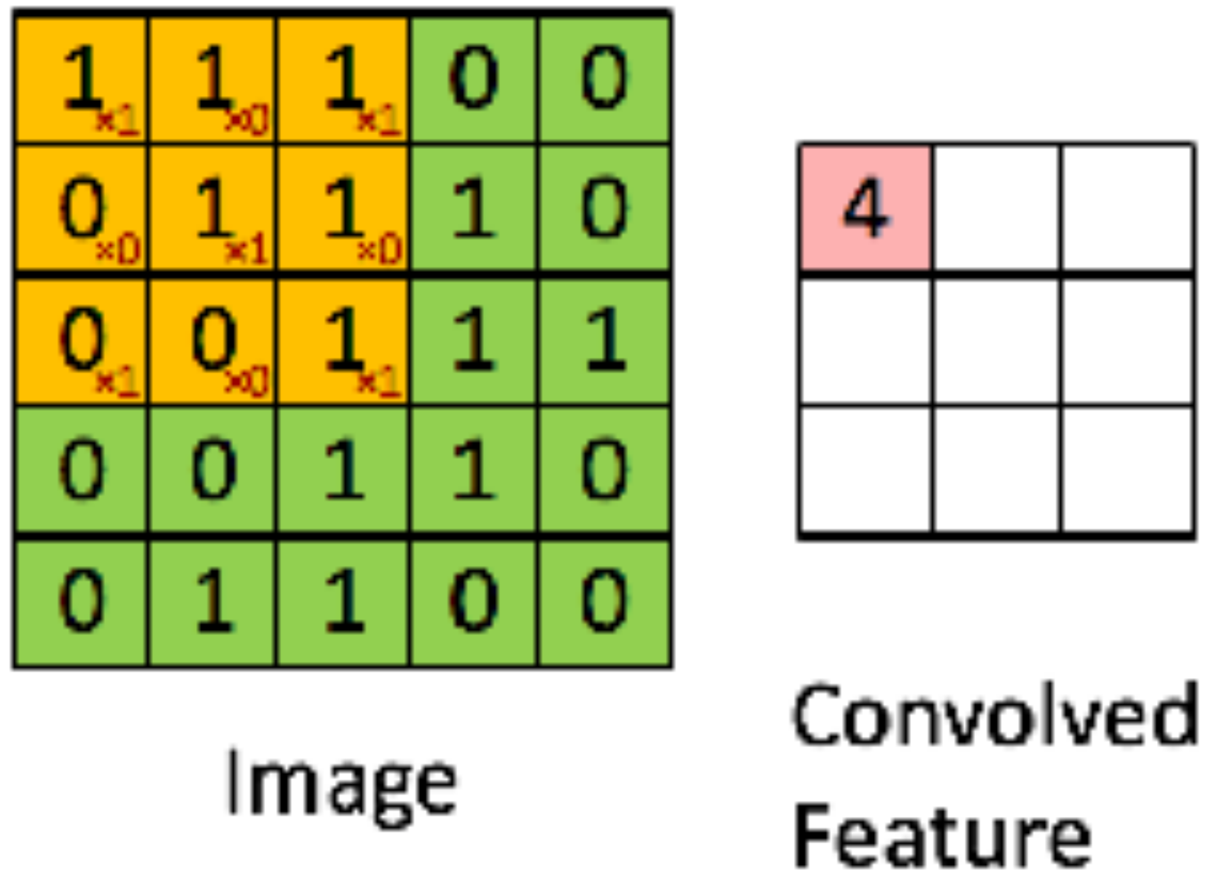
Example: Convolution on an Image

- Suppose you are given the following binary image, X

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

- You want to convolve this image with matrix, W as shown below

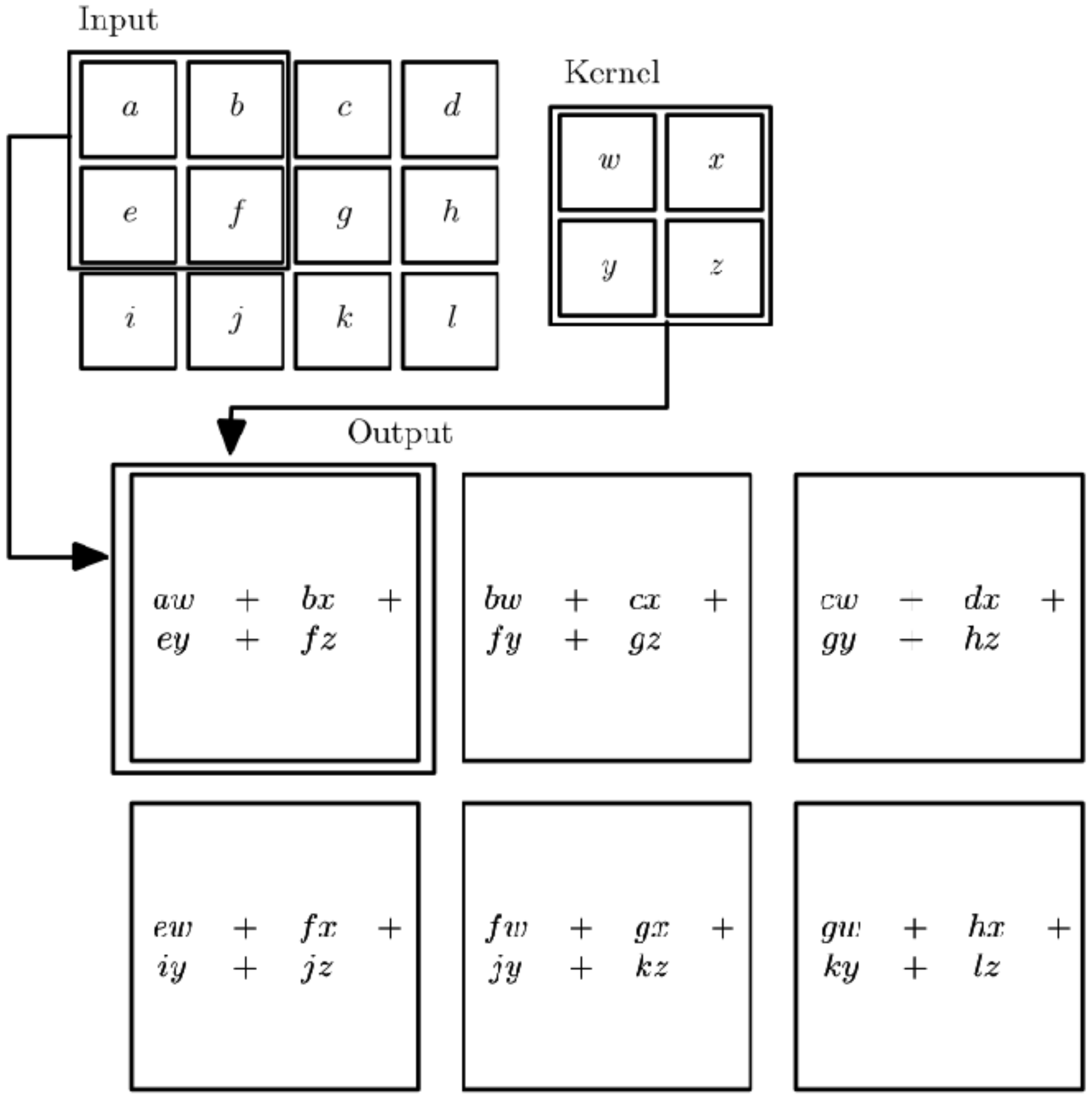
$$
\begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1
\end{bmatrix}
$$



Images courtesy of *the data science blog*

Convolution Operation

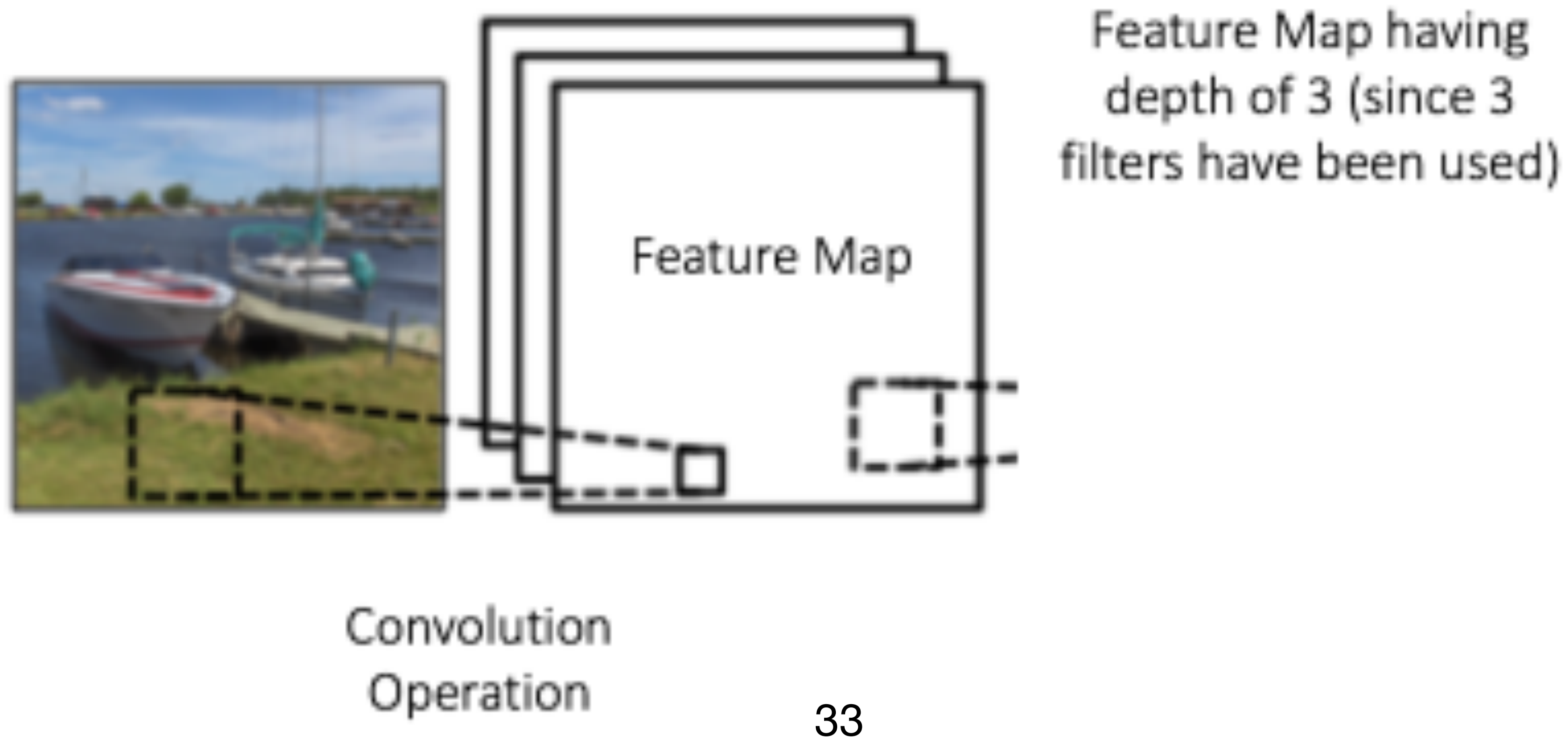
- A mathematical depiction of the convolution operation on an input image
- A CNN learns the values of the filter (or kernel) on its own during the training process



Images courtesy of [(Goodfellow 2016)]

Feature Map

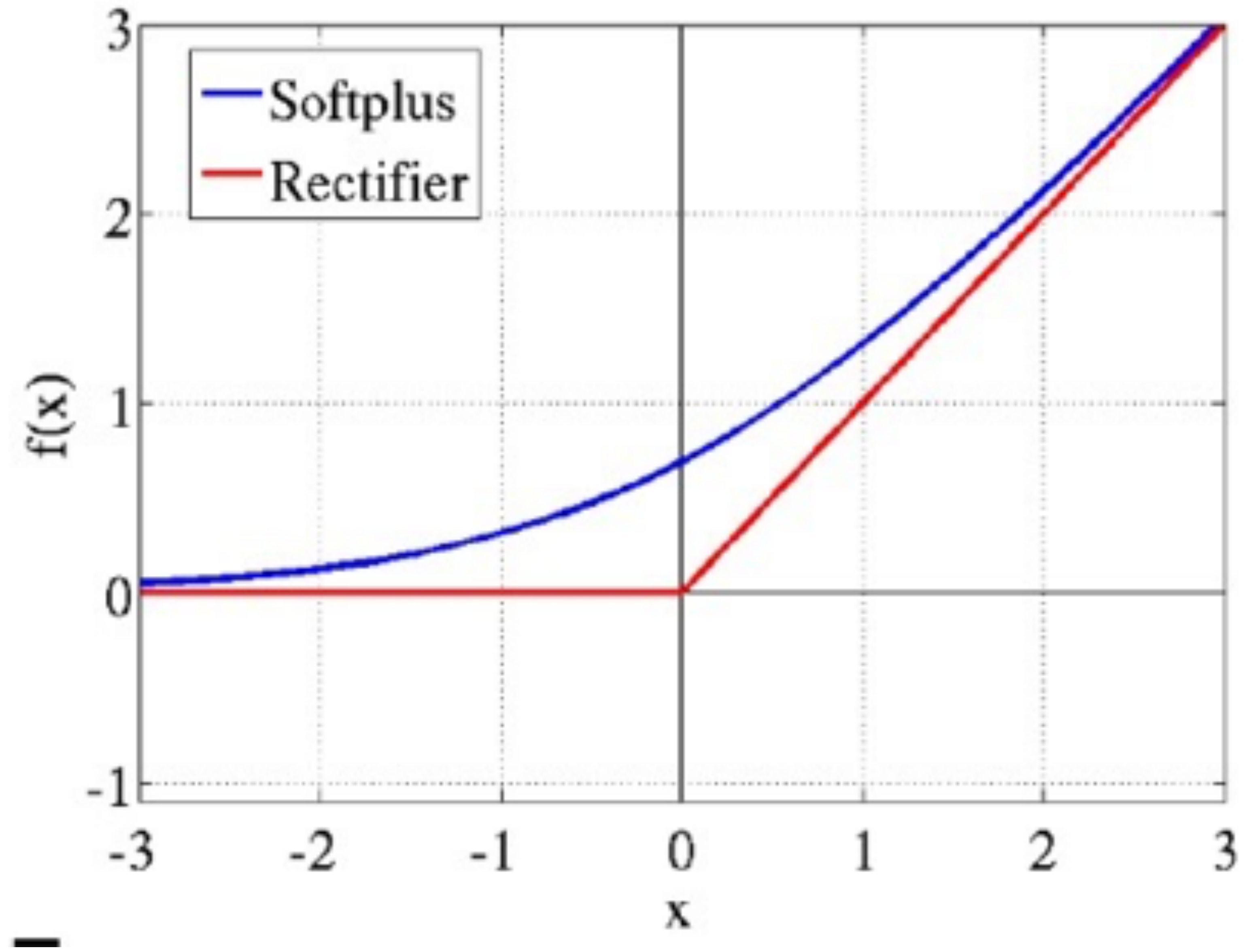
- The size of the Feature Map (resulting image after convolution) is controlled by three parameters
  - **Depth**: Number of different filters to use for the convolution operation
  - **Stride**: Number of pixels used to slide the filter across the input
  - **Zero-padding**: May pad the input with zeros around the border



Images courtesy of *the data science blog*

Rectified Linear (ReLU) Activation

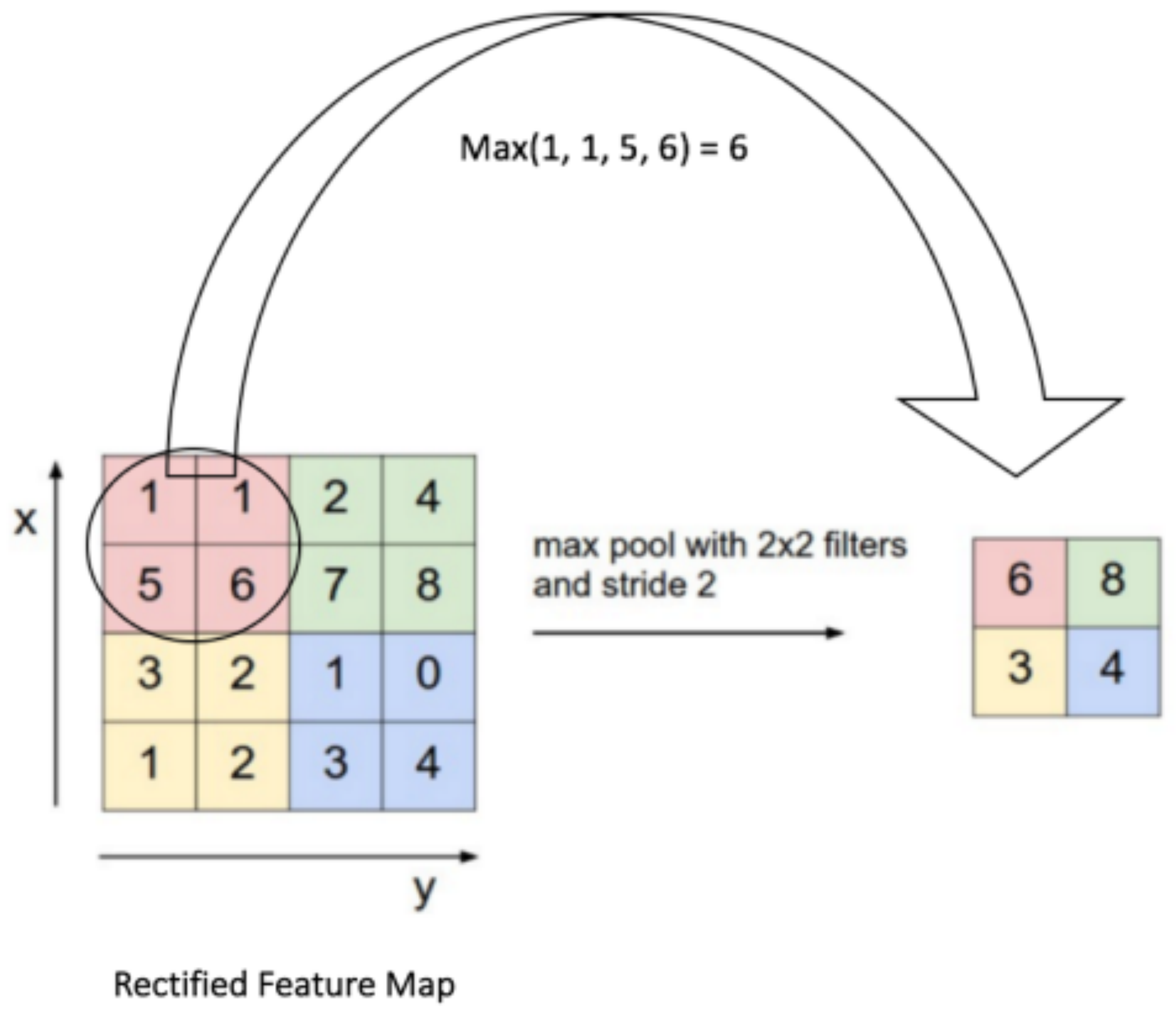
- A rectified linear (ReLU) activation operation may be applied after the convolution operation
  - Introduces nonlinearity to the network
  - ReLU(x) = max(0, x)
  - Applied to every element (pixel)
  - Negative values are replaced by 0



- Other nonlinear activation functions may be used instead

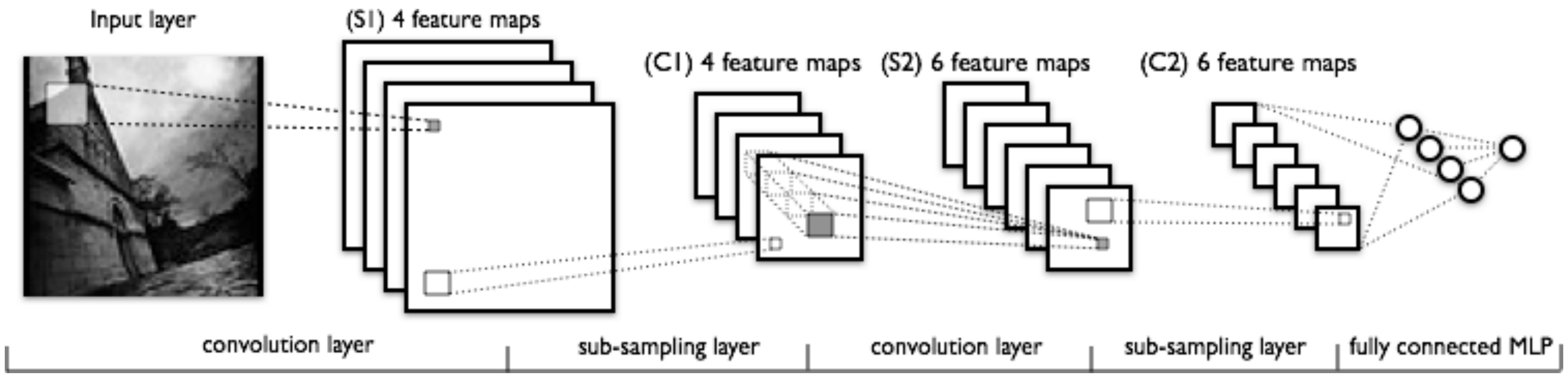
Pooling

- Pooling (aka spatial pooling, subsampling, or downsampling) is used to reduce the dimensionality of the feature map
- Different types of pooling include: Max, Average, Sum, etc.
- A window is defined, and the pooling operation is performed over the elements within that window
- The pooling window slides over the feature map by the stride amount
- It is applied to each feature map



CNN

- Multiple layers of Convolution, Activation, and Pooling may be used in a CNN
- These layers act as feature extraction, to find useful features from the input
- Generally, a final Fully Connected layer is added via a DNN for classification or regression purposes

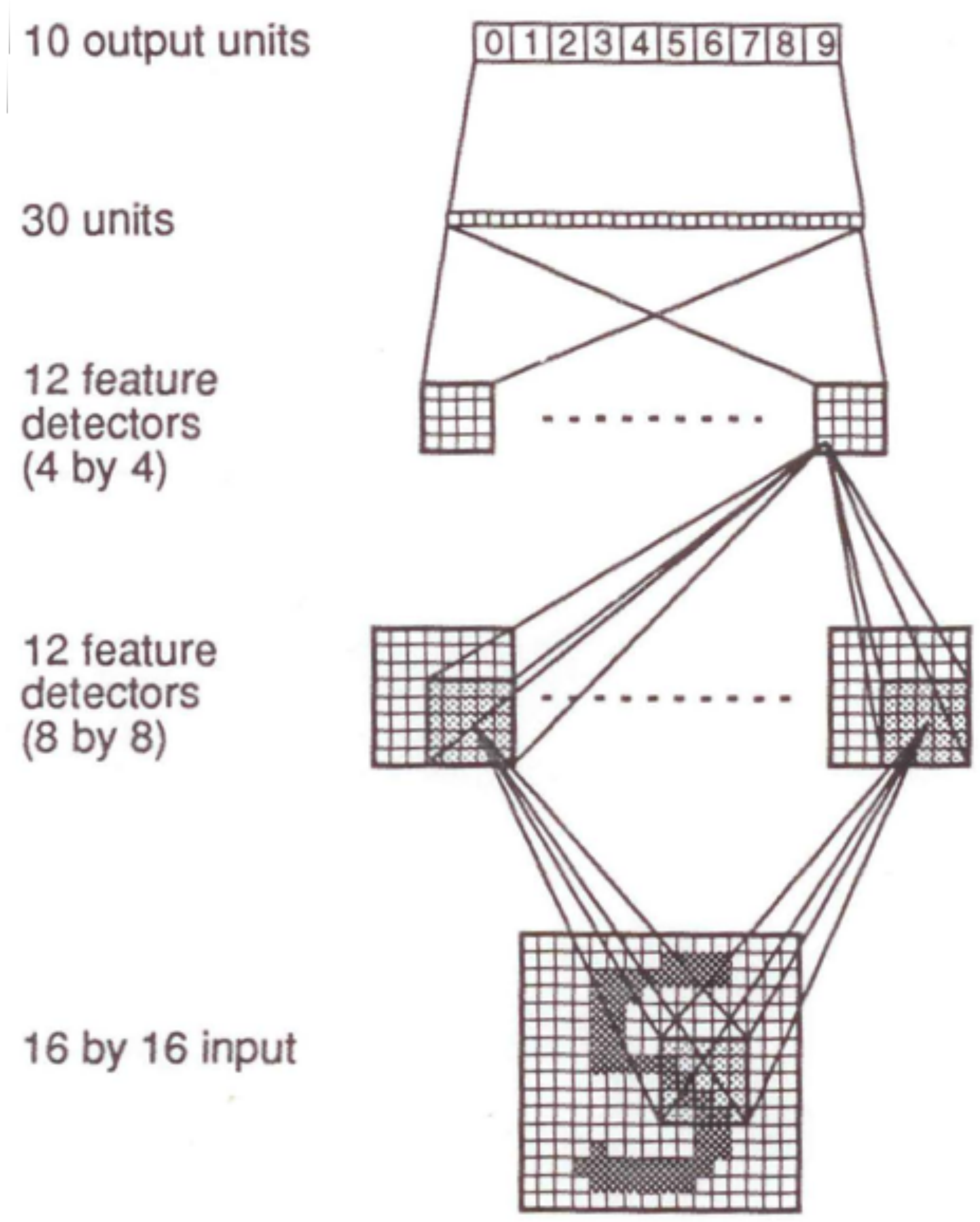


CNN Training

- The Backpropagation algorithm is used to train the parameters of a CNN
- Basic steps:
  - Randomly initialize all filters (or kernels) and weights
  - Propagate the input forward through the layers of the CNN (convolution, activation, pooling, DNN) to get an output(s)
  - Calculate the error between the actual output(s) and the desired output(s)
  - Use Backpropagation to calculate the gradients and deltas, and then update the filters and weights accordingly

An Early CNN Application

- **Task**: Handwritten Zip code Recognition (1989)
- **Network Description**
  - Input: binary pixels for each digit
  - Output: 10 digits
  - Architecture: 4 layers (16 x 16 – 12x8x8 – 12x4x4 -30 -10)
- **Performance**: Trained on 7300 digits and tested on 2000 new ones
  - Achieved 1% error on the training set and 5% error on the test set
  - If allowing rejection (no decision), 1% error on the test set
  - This task is not easy



Next Class

Support Vector Machines