

CS11-711 Advanced NLP

# Recurrent Neural Networks

Sean Welleck

**Carnegie  
Mellon  
University**



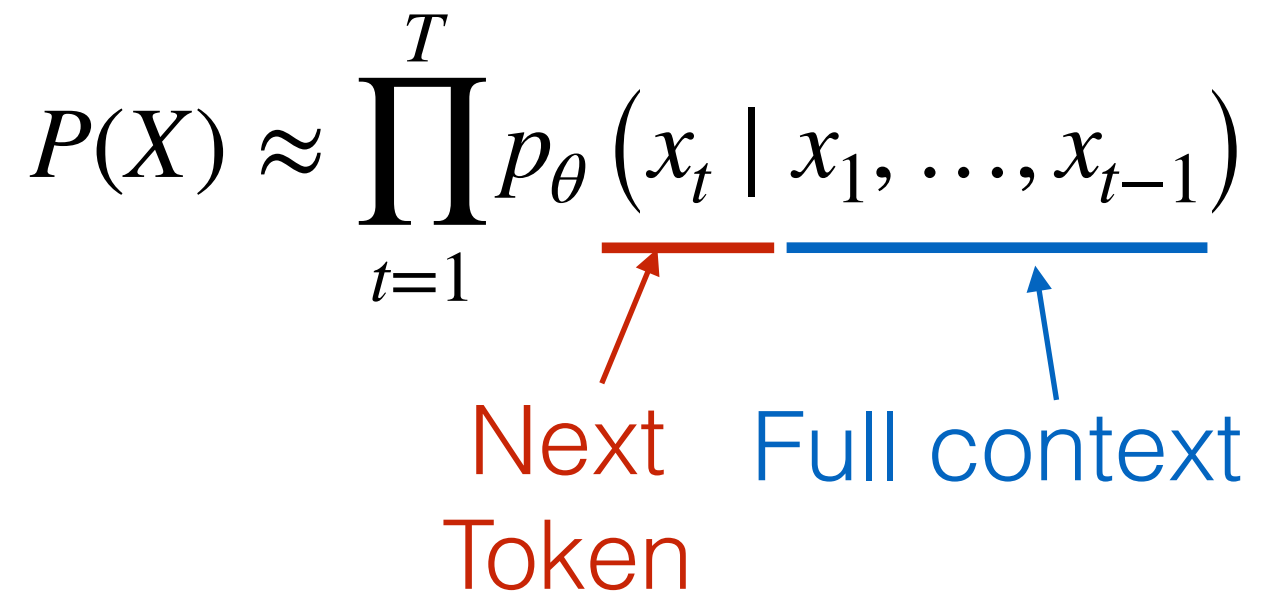
<https://cmu-l3.github.io/anlp-fall2025/>  
<https://github.com/cmu-l3/anlp-fall2025-code>

# Recap

- N-gram models and feedforward architecture
  - Key limitation: a very short context (N-1 tokens)

# This lecture

- Recurrent neural networks
  - In theory, infinite context
  - Motivates *attention*
- Next lecture: attention and transformers

$$P(X) \approx \prod_{t=1}^T p_{\theta} (x_t \mid x_1, \dots, x_{t-1})$$


The diagram shows the equation  $P(X) \approx \prod_{t=1}^T p_{\theta} (x_t \mid x_1, \dots, x_{t-1})$ . A red horizontal line is drawn under the term  $x_t$  in the conditional probability, with a red arrow pointing from the text "Next Token" below it to this line. A blue horizontal line is drawn under the entire sequence  $x_1, \dots, x_{t-1}$  in the conditional probability, with a blue arrow pointing from the text "Full context" below it to this line.

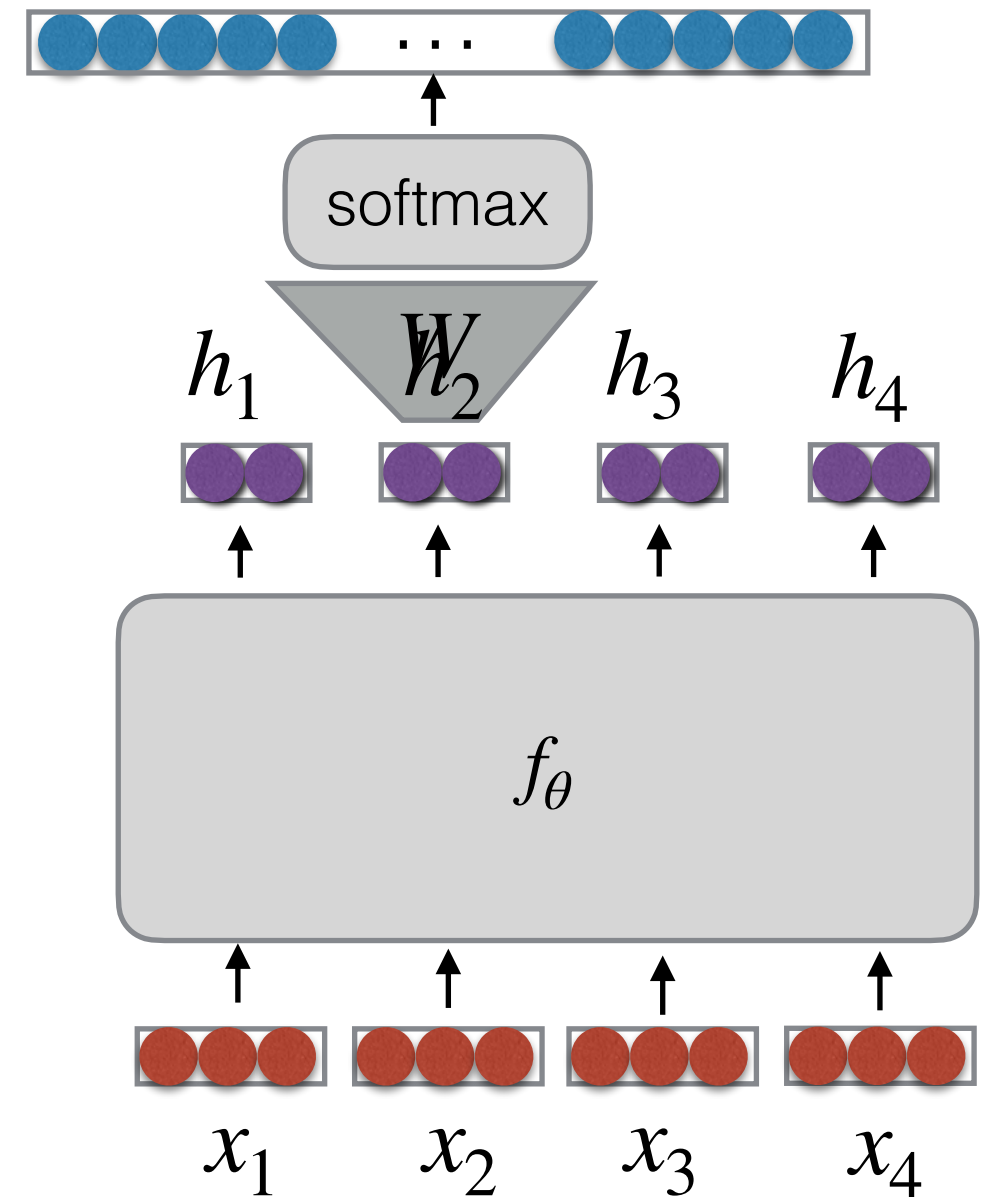
# Outline

- Recurrent neural networks
- Vanishing gradients and other recurrent architectures
- Encoder-decoder
- Attention

# Recurrent Neural Networks

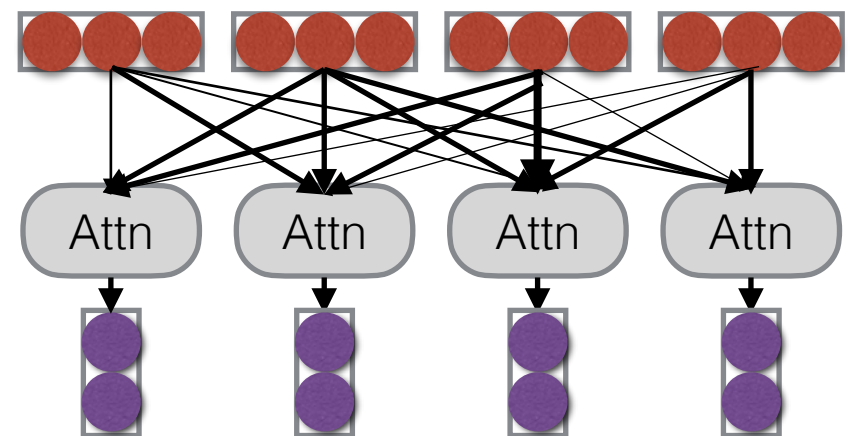
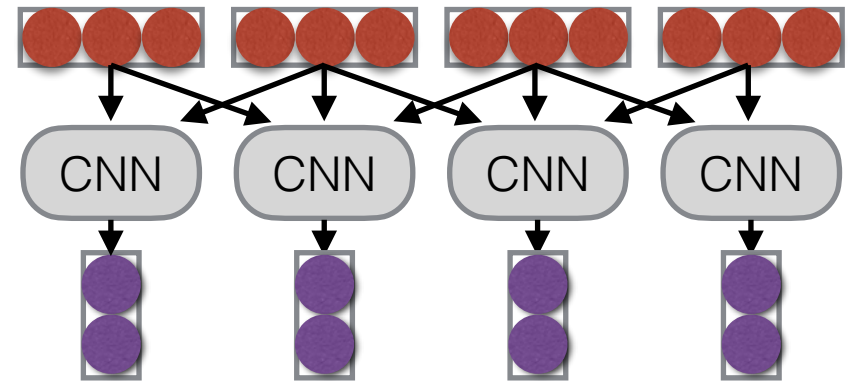
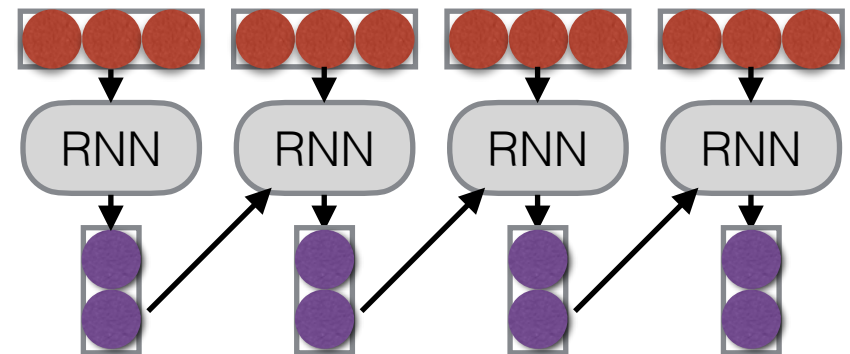
# Sequence model

- $f_{\theta}(x_1, \dots, x_{|x|}) \rightarrow h_1, \dots, h_{|x|}$
- $h_t \in \mathbb{R}^d$ : hidden state
- Example task: language modeling:
- $p_{\theta}(\cdot | x_{<t}) = \text{softmax}(Wh_t^{\top})$



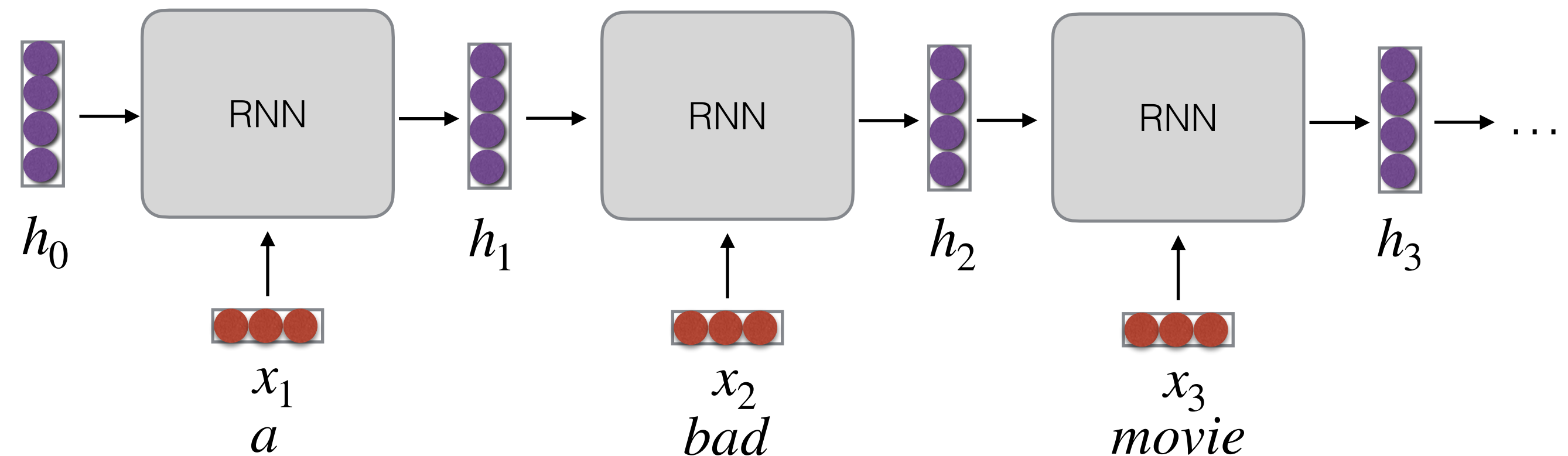
# Three Types of Sequence Models

- **Recurrence:** Condition representations on an encoding of the history
- **Convolution:** Condition representations on local context
- **Attention:** Condition representations on a weighted average of all tokens



# Recurrent neural network

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \quad \sigma : \text{activation function (tanh, relu, ...)}$$



Parameters  $\theta$

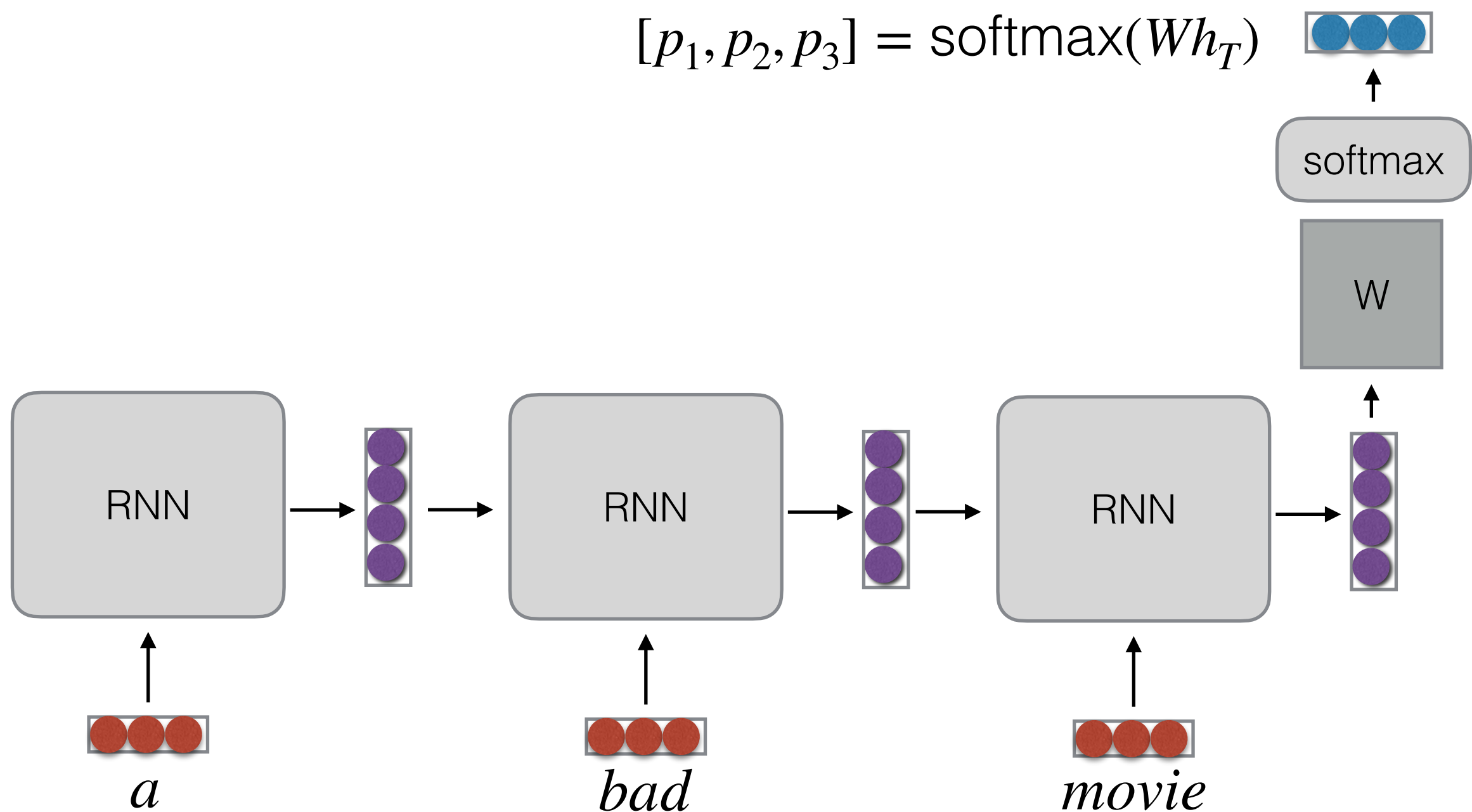
$$W_h \in \mathbb{R}^{d \times d}$$
$$W_x \in \mathbb{R}^{d \times d_{in}}$$
$$b \in \mathbb{R}^d$$



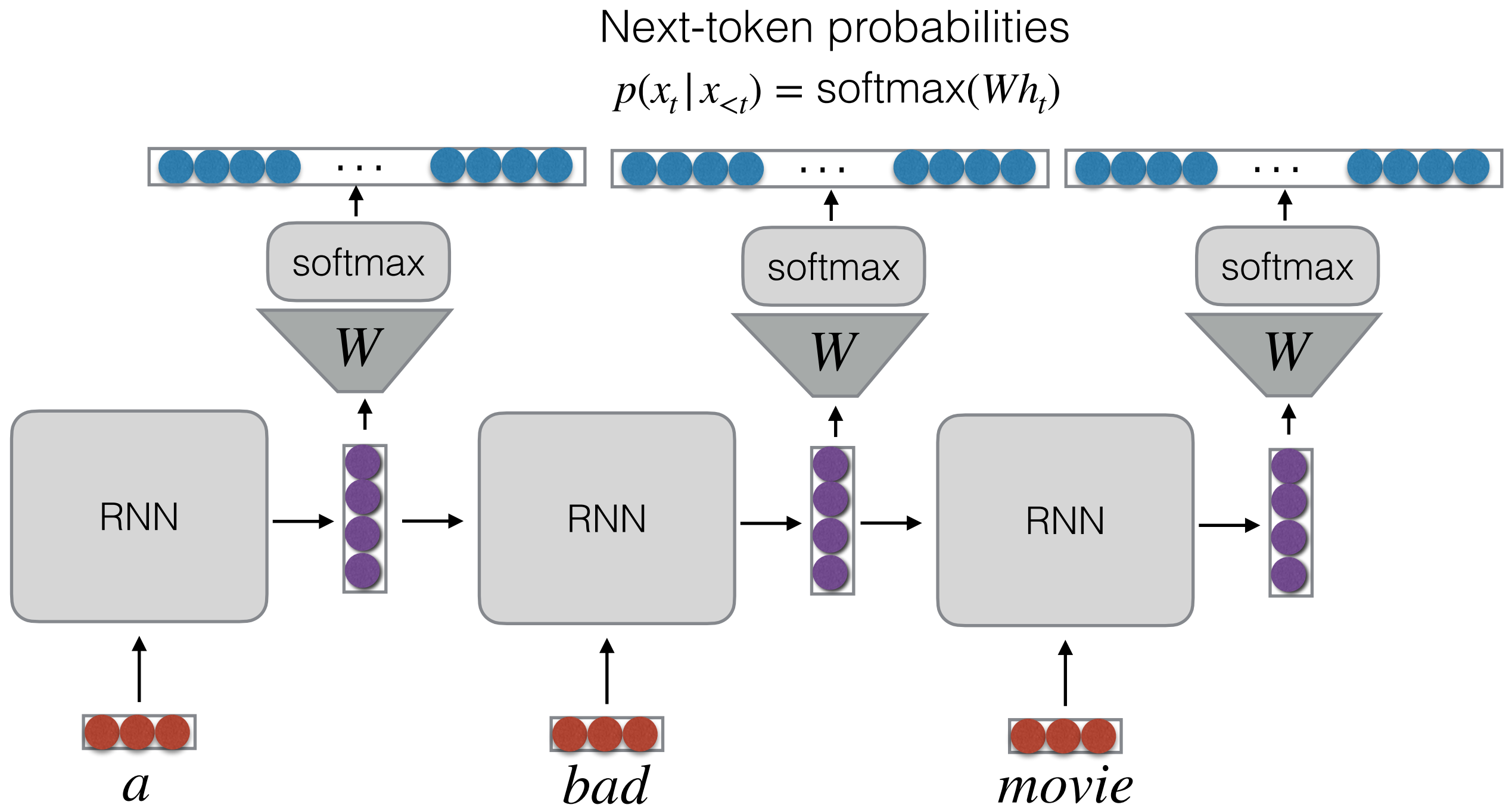
# Example: sequence classification

Output class probabilities

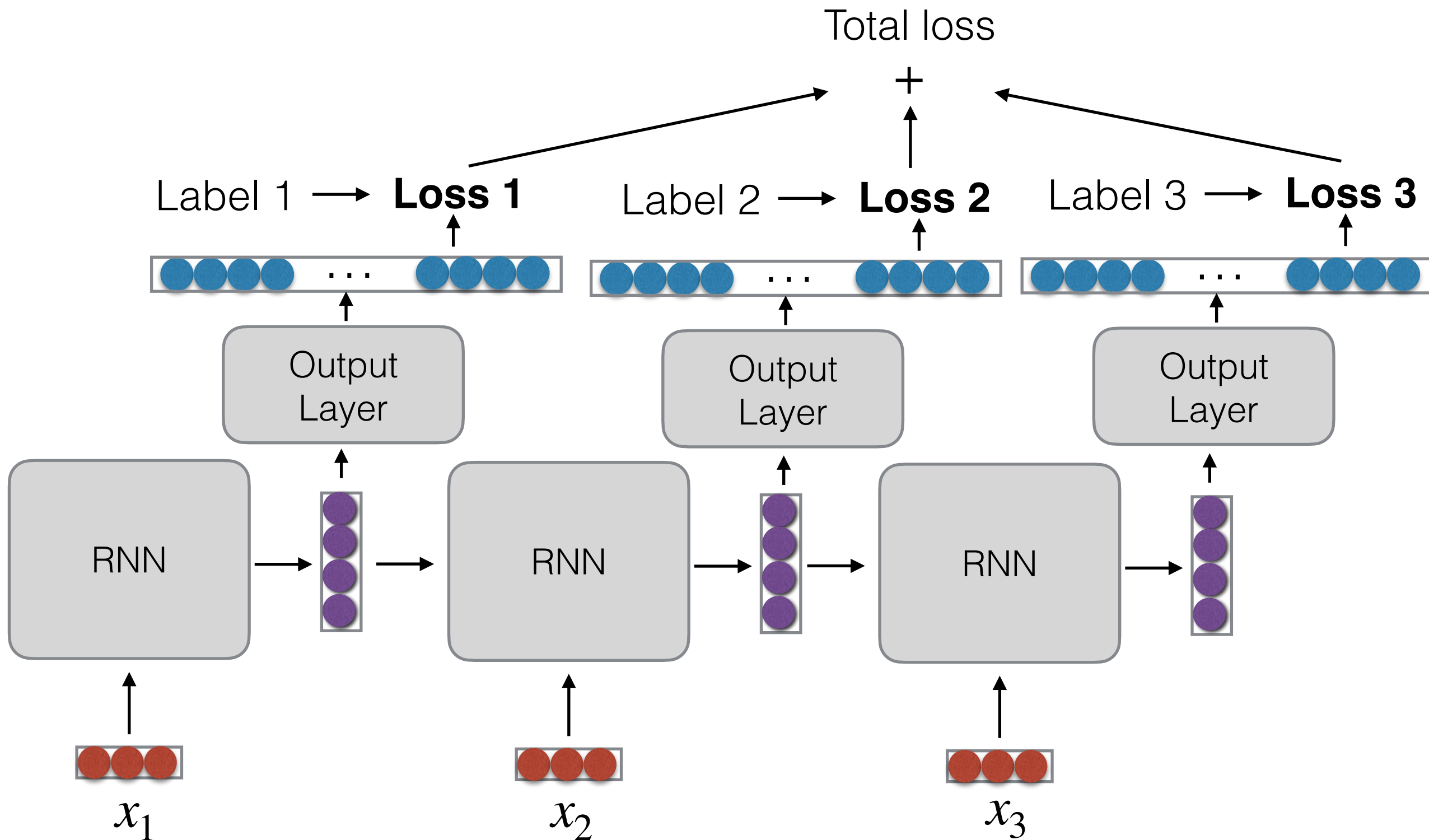
$$[p_1, p_2, p_3] = \text{softmax}(Wh_T)$$



# Example: language modeling

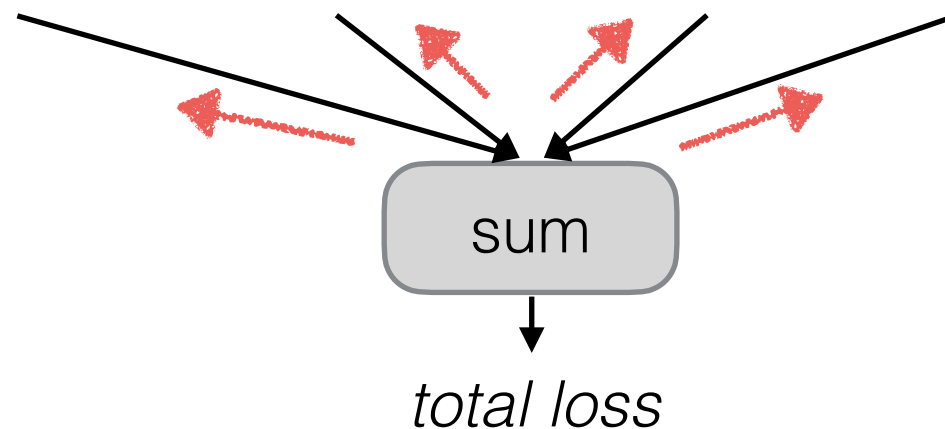


# Training RNNs



# RNN Training

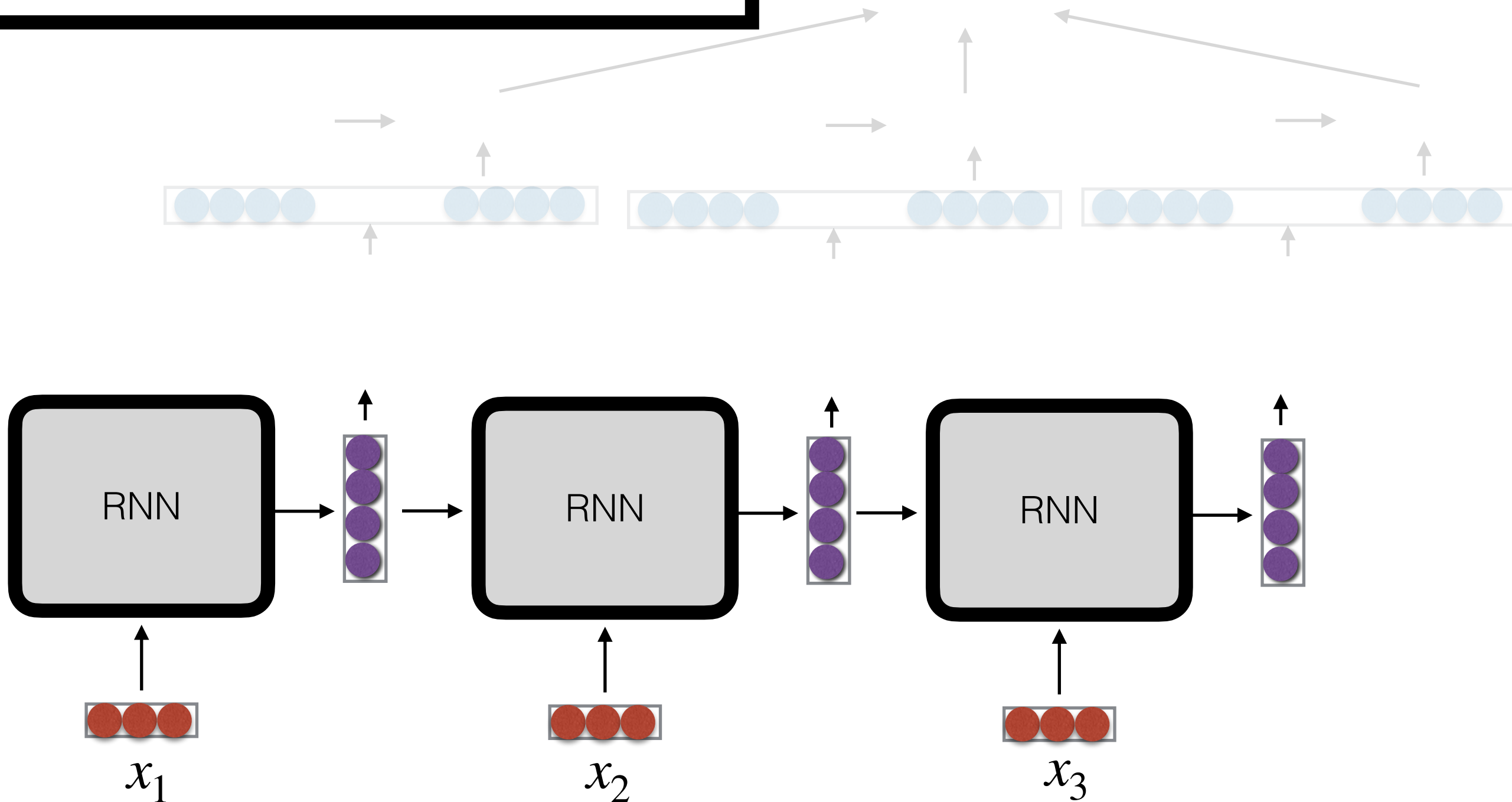
- The unrolled graph is a well-formed (DAG) computation graph—we can run backpropagation



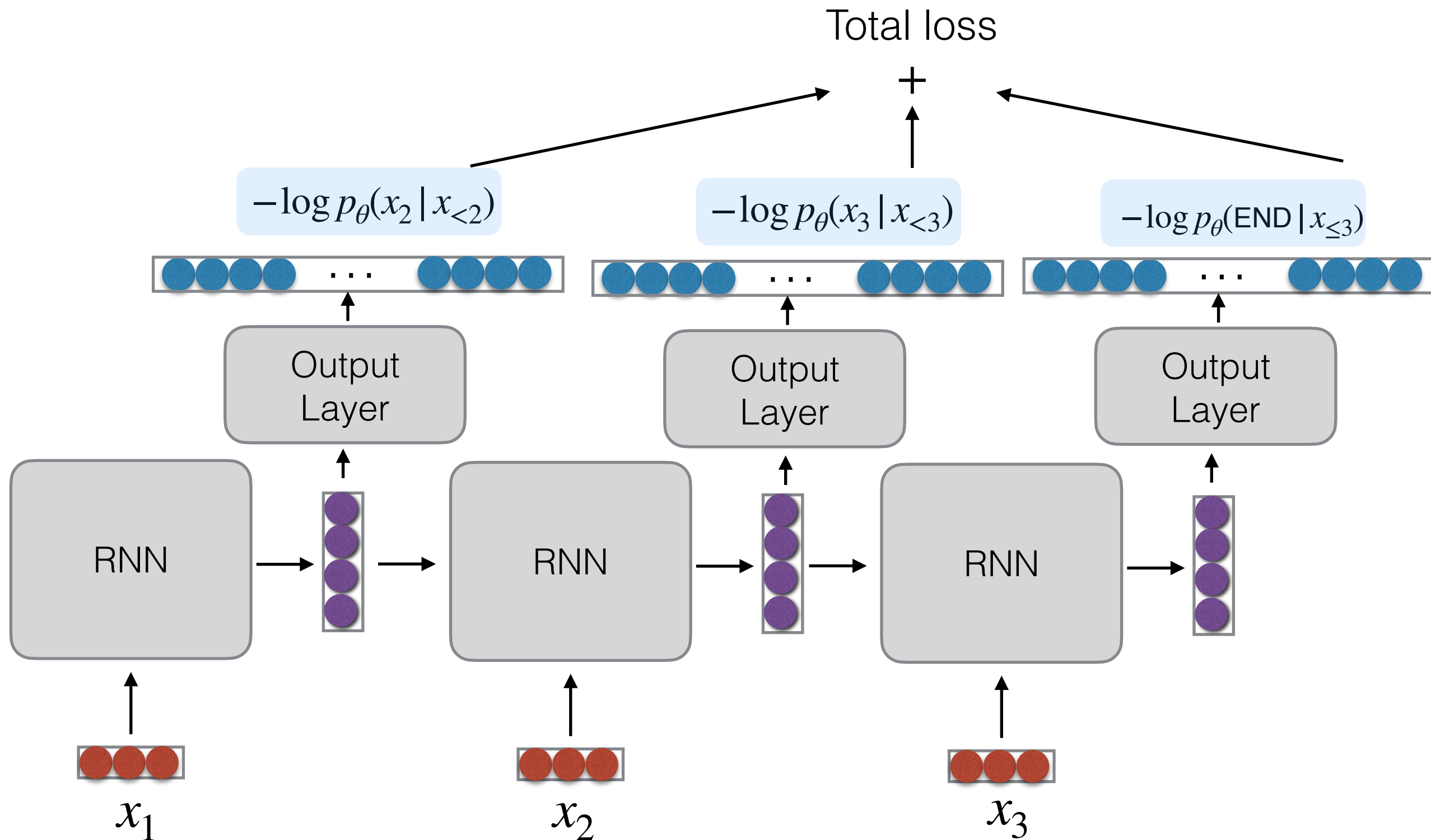
- This is historically called “backpropagation through time” (BPTT)

# Parameter tying

Same parameters; gradients  
are accumulated



# Training RNNs Example: Language Modeling



# Training RNNs: Language Modeling

- Maximum likelihood estimation (again!)

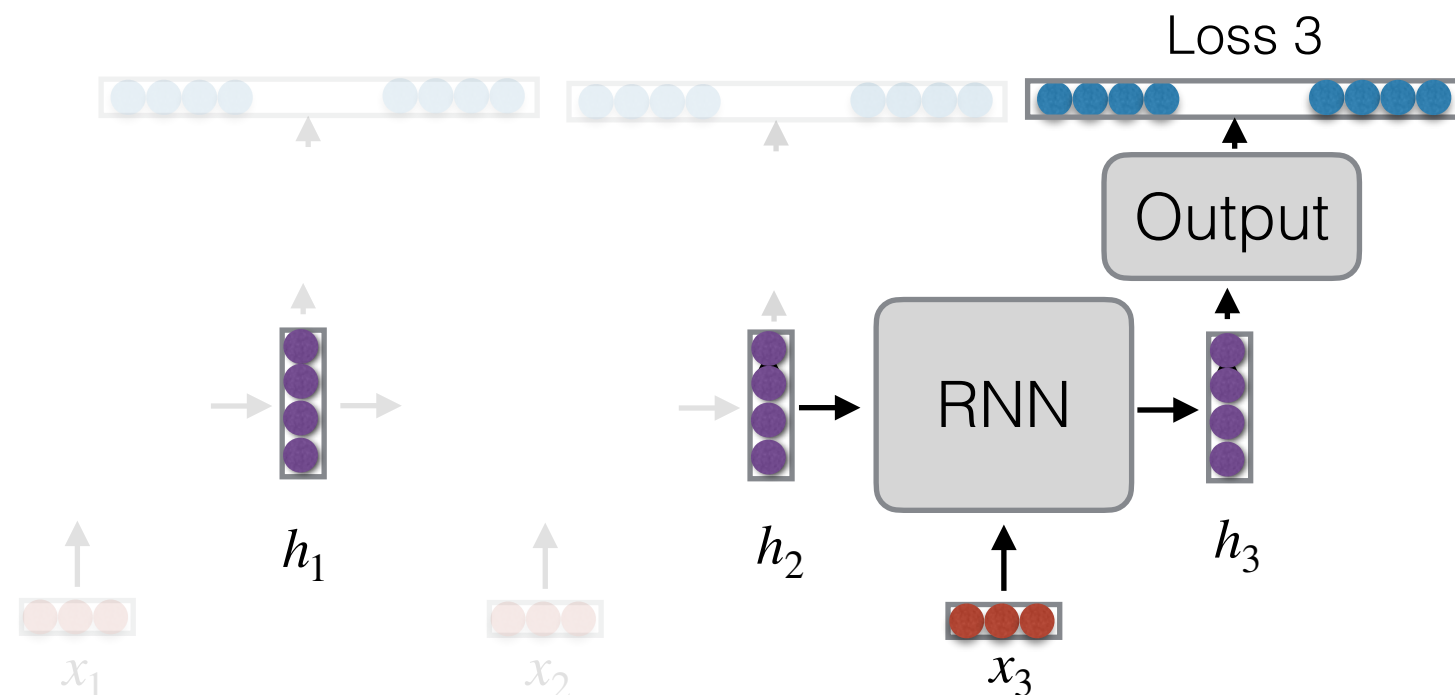
$$\bullet \max \sum_{x \in D_{train}} \log p_{\theta}(x)$$

$$\equiv \min - \sum_{x \in D_{train}} \sum_t \log p_{\theta}(x_t | x_{<t})$$

Previous slide

# Training RNNs: Language Modeling

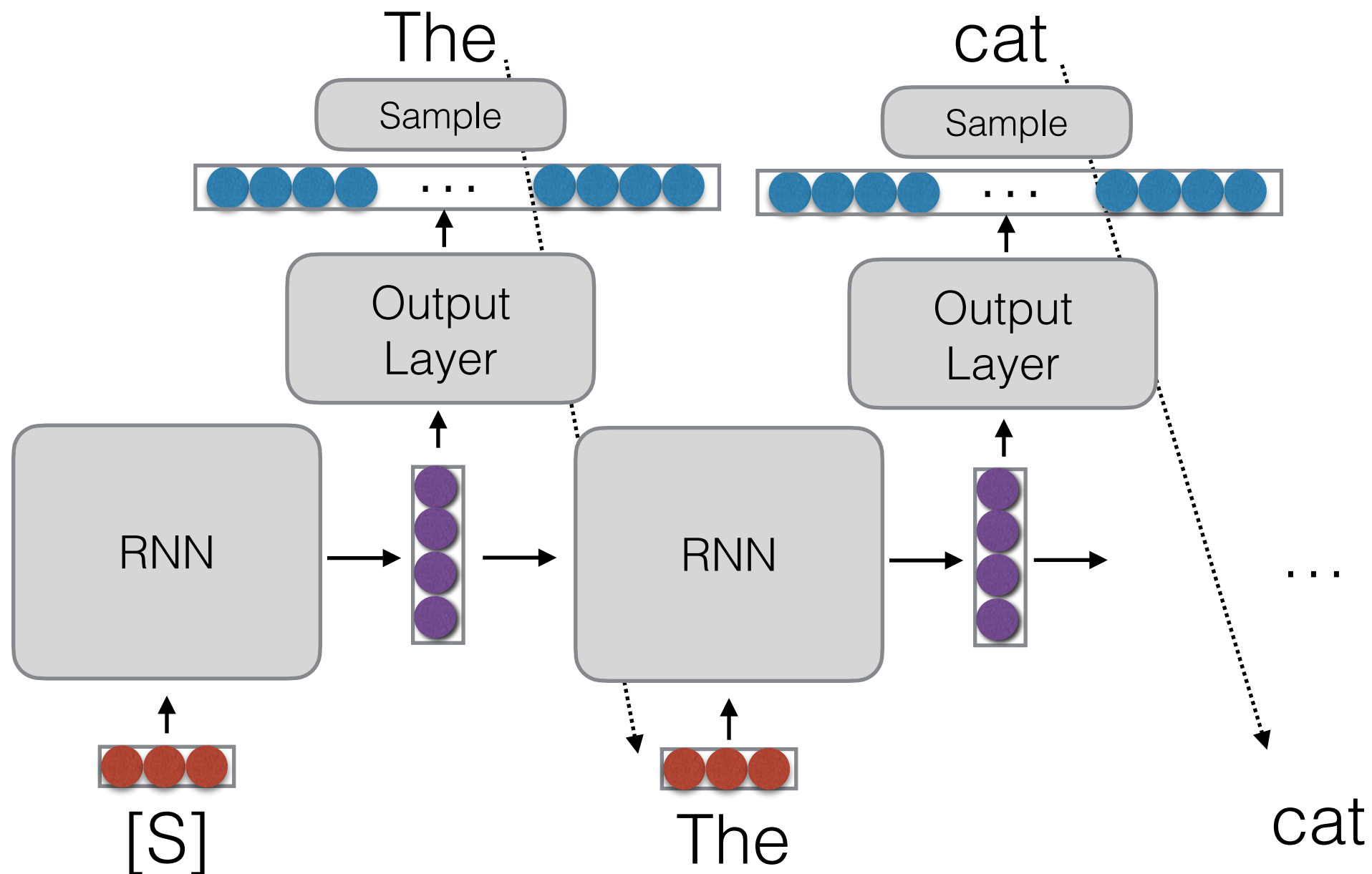
- Computing the loss at step  $t$  requires computing the hidden state  $h_t$
- Computing  $h_t$  requires  $h_{t-1}, h_{t-2}, \dots$
- As a result, RNN training is **difficult to parallelize**





# RNN Inference: Language Models

- Generate one token, use the new hidden state for the next step, repeat



# RNN Inference: Language Models

- We only need to store the previous hidden state
  - Constant memory as sequence length increases
- Each step is a “local” computation,  $O(1)$ 
  - $O(T)$  computation for a length  $T$  sequence

# Recap: RNNs

- A sequence model,  $f_{\theta}(x_1, \dots, x_{|x|}) \rightarrow h_1, \dots, h_{|x|}$
- Transforms a *hidden state* at each step
  - $h_t = \sigma(W_h h_{t-1} + W_x x_t + b)$
  - Intuitively, the hidden state is a “memory” mechanism
- We can use it for tasks such as language modeling, and train it with backpropagation
- Recurrent hidden state makes parallelization difficult

# In Code

```
class RNNCell(torch.nn.Module):  
    def __init__(self, input_size, hidden_size):  
        super(RNNCell, self).__init__()  
        self.input_size = input_size  
        self.hidden_size = hidden_size  
        self.Wh = torch.nn.Linear(hidden_size, hidden_size)  
        self.Wx = torch.nn.Linear(input_size, hidden_size)  
        self.activation = torch.nn.Tanh()  
  
    def forward(self, x, h):  
        h = self.activation(self.Wh(h) + self.Wx(x))  
        return h
```

# In Code

```
class RNNLM(nn.Module):
    def __init__(self, vocab_size, hidden_size):
        super(RNNLM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.rnn = RNNCell(hidden_size, hidden_size)
        self.output = nn.Linear(hidden_size, vocab_size)
        self.hidden_size = hidden_size

    def forward(self, x, hidden=None):
        if hidden is None:
            hidden = self.init_hidden(x.size(0))

        x = self.embedding(x)

        outs = []
        for i in range(x.size(1)):
            hidden = self.rnn(x[:, i:i+1], hidden)
            out = self.output(hidden)
            outs.append(out)

        outs = torch.cat(outs, dim=1)
        return outs, hidden

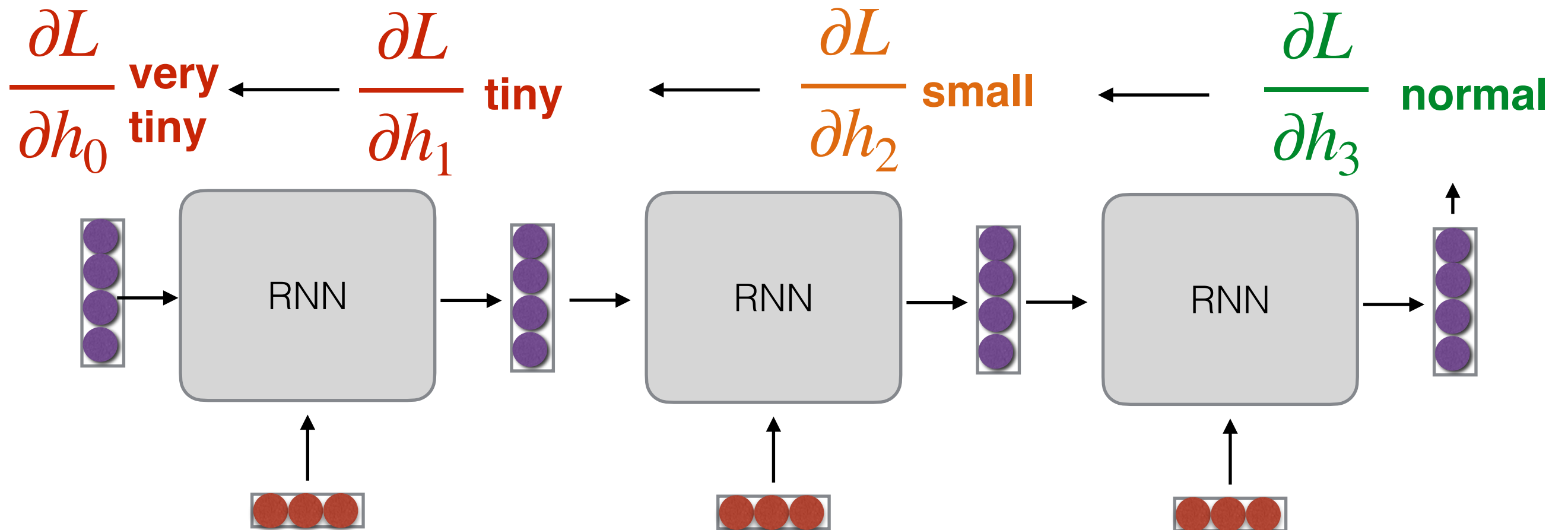
    def init_hidden(self, batch_size):
        return torch.zeros(batch_size, 1, self.hidden_size)
```

# Outline

- Recurrent neural networks
- **Vanishing gradients and other recurrent architectures**
- Encoder-decoder
- Attention

# Vanishing Gradients

# Vanishing gradient



- Gradients decrease as they get pushed back
- **Implication:** Cannot model long dependencies!



# Vanishing gradient: why?

Normal RNN:  $h_t = \tanh(W_{in}x + Wh_t)$ ,  $y_T = W_{out}h_T$

$$\frac{\partial L}{\partial W} = \sum_{t=0}^T \frac{\partial L}{\partial y_T} \frac{\partial y_T}{\partial h_T} \frac{\partial h_T}{\partial h_t} \frac{\partial h_t}{\partial W}$$

$$\frac{\partial h_T}{\partial h_t} = \frac{h_T}{h_{T-1}} \frac{\partial h_{T-1}}{\partial h_{T-2}} \dots \frac{\partial h_{t+1}}{\partial h_t} = \prod_{t'=t}^T \frac{\partial h_{t'+1}}{\partial h_{t'}}$$

$$\frac{\partial h_{t'+1}}{\partial h_{t'}} = \text{diag} \left( \tanh'(W_{in}x_{t'+1} + Wh_{t'}) \right) W$$

Derivative of tanh is in  $[0, 1]$

$$W = VDV^{-1}: \text{when dominant eigenvalue} < 1, D^{T-t} \rightarrow 0$$

# A solution: gating and additive connections

- **Basic idea:** pass information across timesteps with a learned “gate”  $z_t = \sigma(W_{zx}x + W_{zh}h_{t-1})$ 
  - $h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t$
- To retain a long-term dependency, the model can set  $z \rightarrow 0$  for multiple steps:

$$\bullet \quad \frac{\partial h_{t_2}}{\partial h_{t_1}} = \prod_{t=t_1}^{t_2} \underbrace{\frac{\partial h_t}{\partial h_{t-1}} h_{t-1}}_1 = 1$$

# A solution: gating and additive connections

- **Basic idea:** pass information across timesteps with a learned “gate”  $z_t$

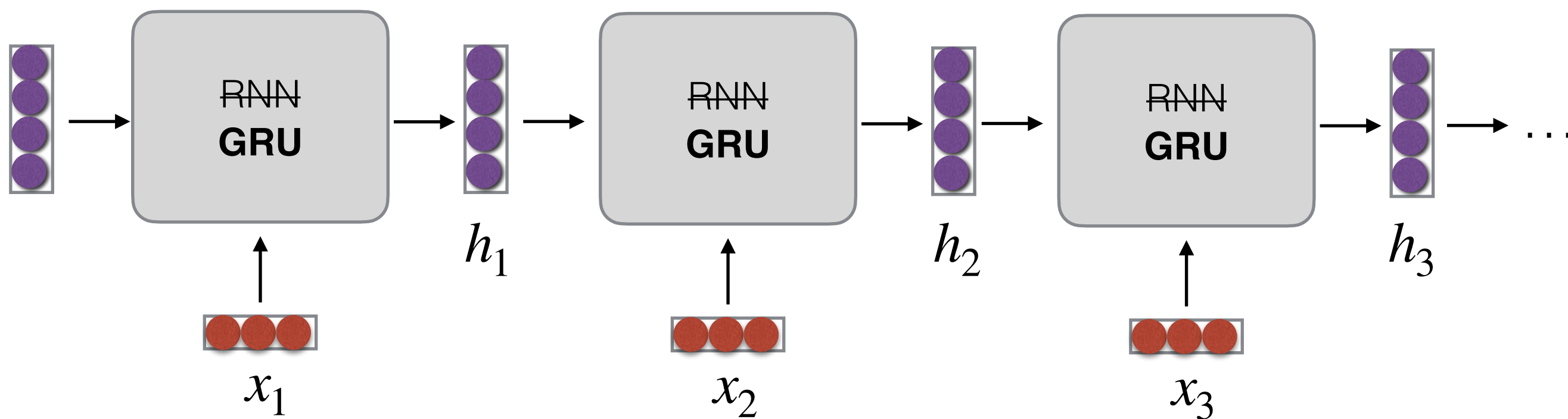
- $$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t$$

- When  $z > 0$ , incorporate a new hidden state  $\tilde{h}_t$ ,  
e.g. similar to a normal RNN

# A solution: gating and additive connections

- No gate: learn the difference  $\tilde{h}_t$  (“residual”)
  - $h_t = h_{t-1} + \tilde{h}_t$

Putting it all together:  
*Gated Recurrent Unit (GRU)*



# Putting it all together: *Gated Recurrent Unit (GRU)*

- “Update gate”

$$z_t = \sigma (W_z x_t + U_z h_{t-1})$$

- “Reset gate”

$$r_t = \sigma (W_r x_t + U_r h_{t-1})$$

- Recurrent update:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t$$

- $\hat{h}_t$  is a “candidate state”

$$\hat{h}_t = \tanh (W_h x_t + U_h (r_t \odot h_{t-1}))$$

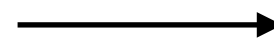
# Putting it all together: gated architectures

- **Gated recurrent unit (GRU)** [Cho et al 2014]:
  - 2 gate architecture
    - Gate 1 (*update*): should I update the previous hidden state?
    - Gate 2 (*reset*): should I use the hidden state in the update?
- **Long short term memory (LSTM)** [Hochreiter & Schmidhuber 1997]:
  - 4 gate architecture using an additional *context* vector
    - Gate 1: should I update the previous context?
    - Other gates: how should I update?

# Recap: vanishing gradients

- Basic RNN: gradients vanish, so we can't model long dependencies in practice
- Better recurrent models help overcome this
  - E.g., GRU, LSTM
- In practice, a drop-in replacement

```
class RecurrentLM(nn.Module):  
    def __init__(self, vocab_size, embedding_size, hidden_size):  
        super(RecurrentLM, self).__init__()  
        self.embedding = nn.Embedding(vocab_size, embedding_size)  
        self.rnn = nn.RNN(embedding_size, hidden_size)  
        self.output = nn.Linear(hidden_size, vocab_size)  
        self.hidden_size = hidden_size
```



```
class RecurrentLM(nn.Module):  
    def __init__(self, vocab_size, em  
        super(RecurrentLM, self).__in  
        self.embedding = nn.Embedding  
        self.rnn = nn.GRU(embedding_s  
        self.output = nn.Linear(hidde  
        self.hidden_size = hidden_siz
```



# Outline

- Recurrent neural networks
- Vanishing gradients and other recurrent architectures
- **Encoder-decoder**
- Attention

Encoder-decoder

# Encoder-decoder

- Motivation: conditional generation

$$p_{\theta}(y_1, \dots, y_T | x)$$

Japanese sentence

Response

...

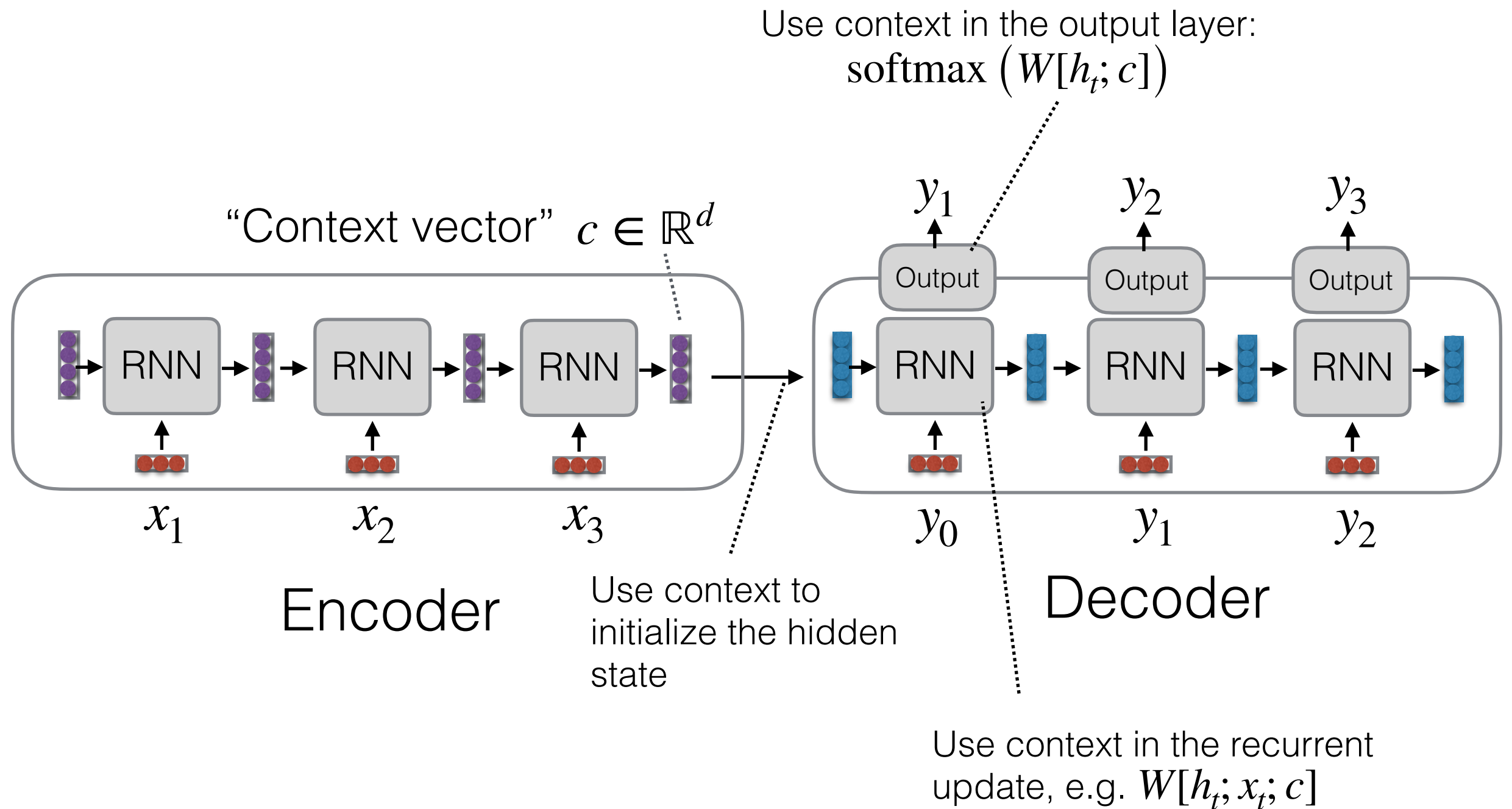
English sentence

Chat history

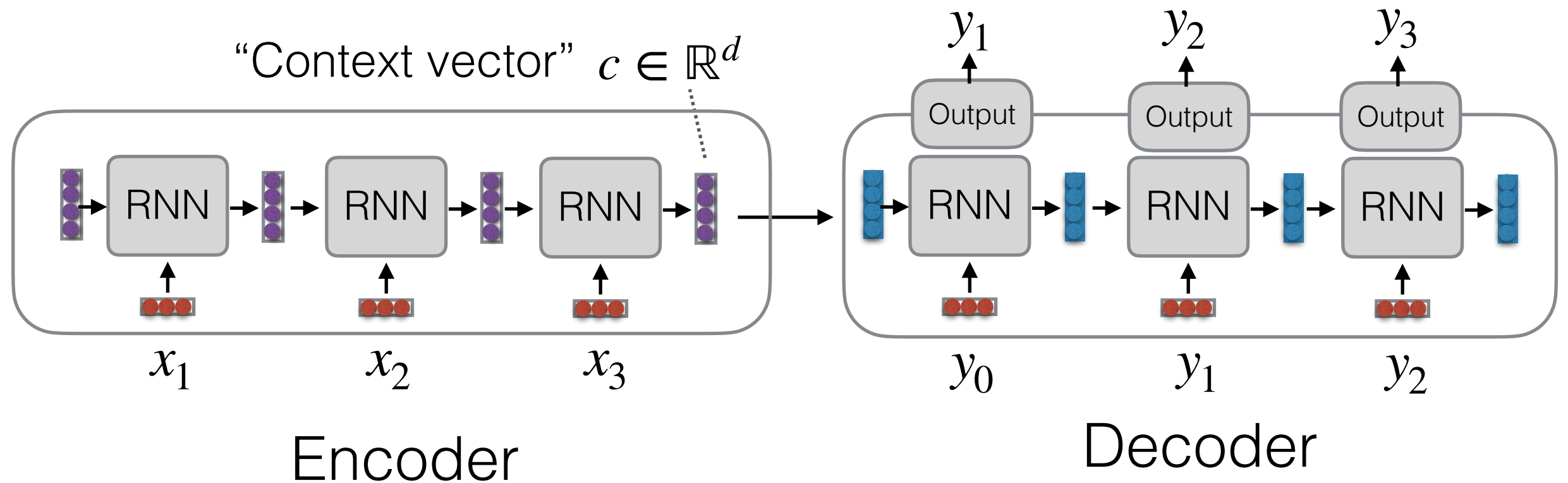
...

- Basic idea: use a sequence model to represent  $x$  as a vector

# Encoder-decoder



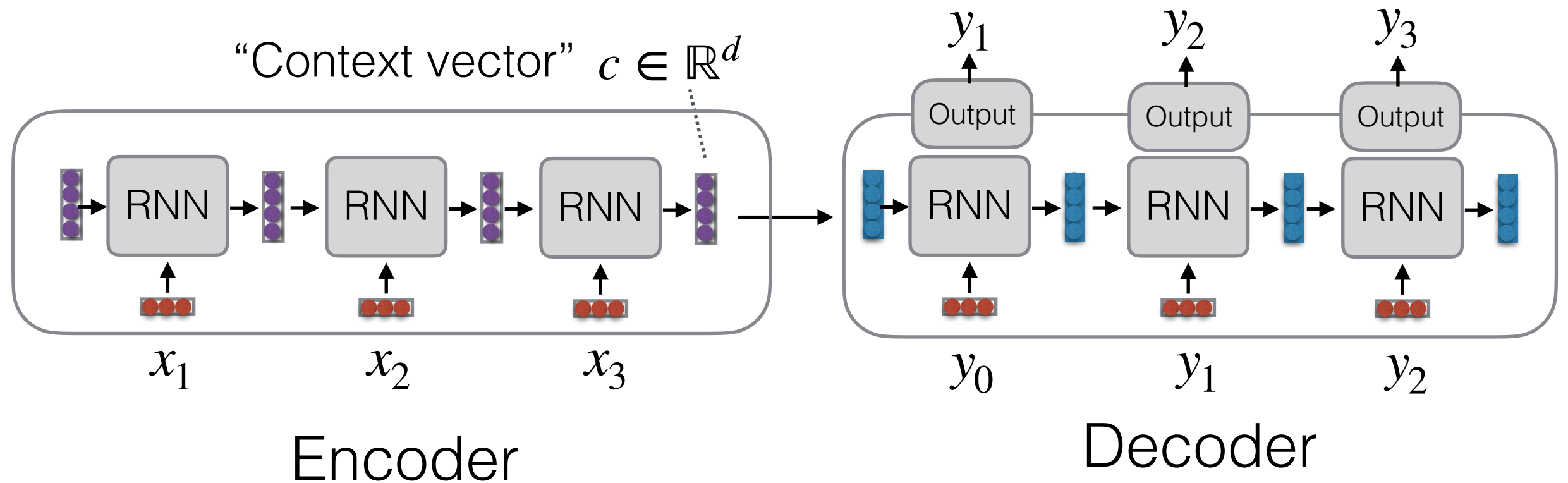
# Encoder-decoder



Training:

$$\min_{\theta} \sum_{(x,y) \in D} \sum_t -\log p_{\theta}(y_t | y_{<t}, x)$$

# Encoder-decoder



*A single context vector is used for all tokens:  
can we do better?*

Attention

# Basic Idea

(Bahdanau et al. 2015)

- Encode each token in the sequence into a vector
- When decoding, perform a linear combination of these vectors, weighted by “attention weights”



# Attention

- **Keys:** Encoder states  $h_1^{enc}, \dots, h_N^{enc}$
- **Query:** Current decoder hidden state  $h$

- Compute attention scores

- $\alpha_n = \text{score}(h, h_n^{enc}) /$

- Output: a weighted sum

- $c = \sum_{n=1}^N \alpha_n h_n^{enc}$

Dot product

$$\text{score}(q, k) = q^T k$$

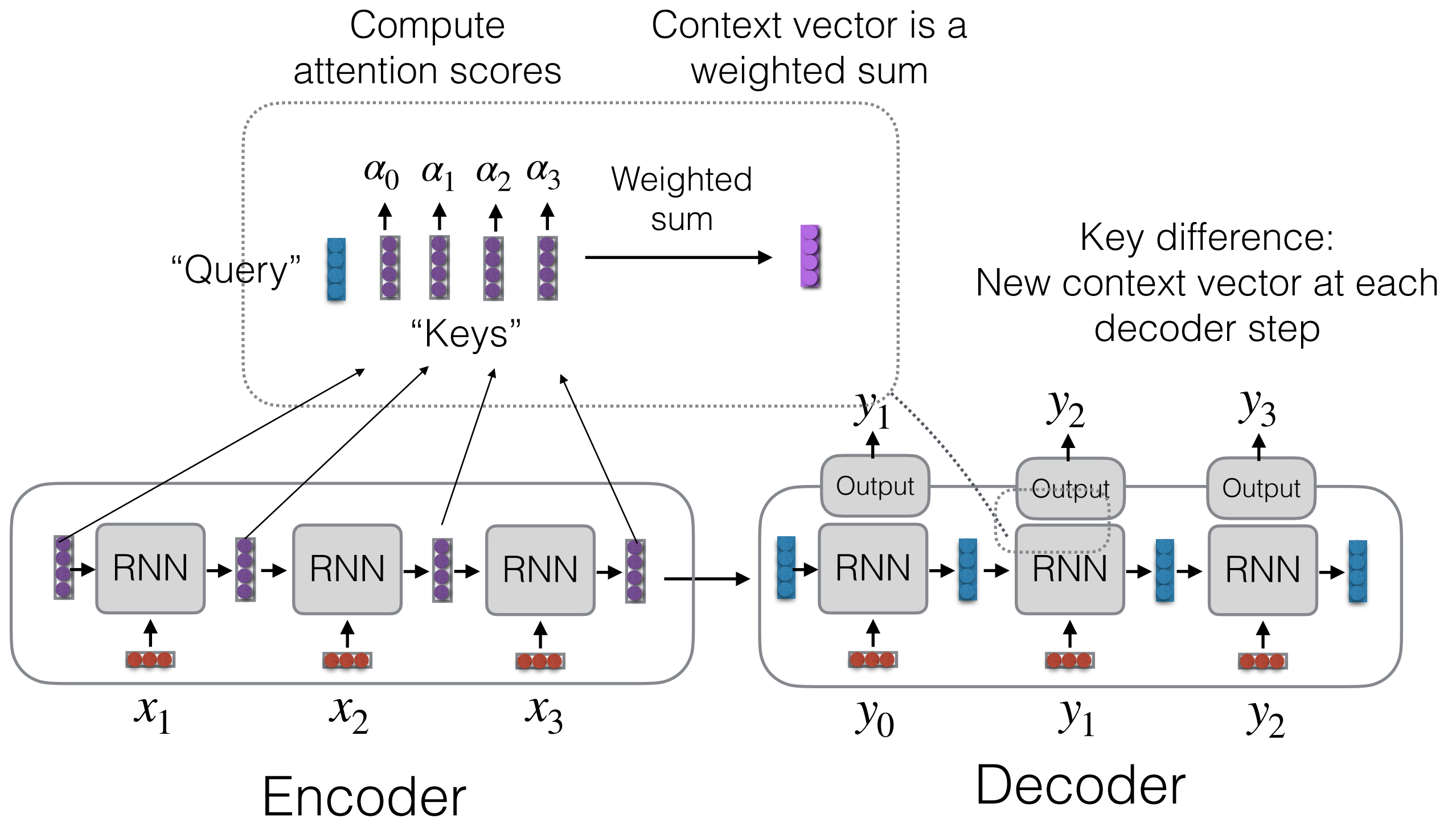
Bilinear

$$\text{score}(q, k) = q W k$$

Nonlinear

$$\text{score}(q, k) = w^T \tanh(W[q; k])$$

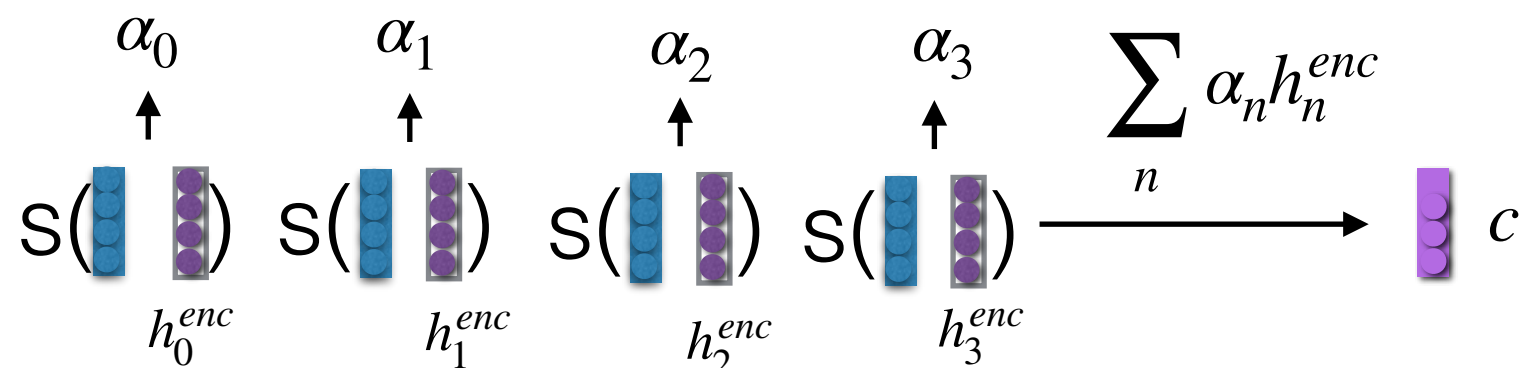
# Attention



# Attention

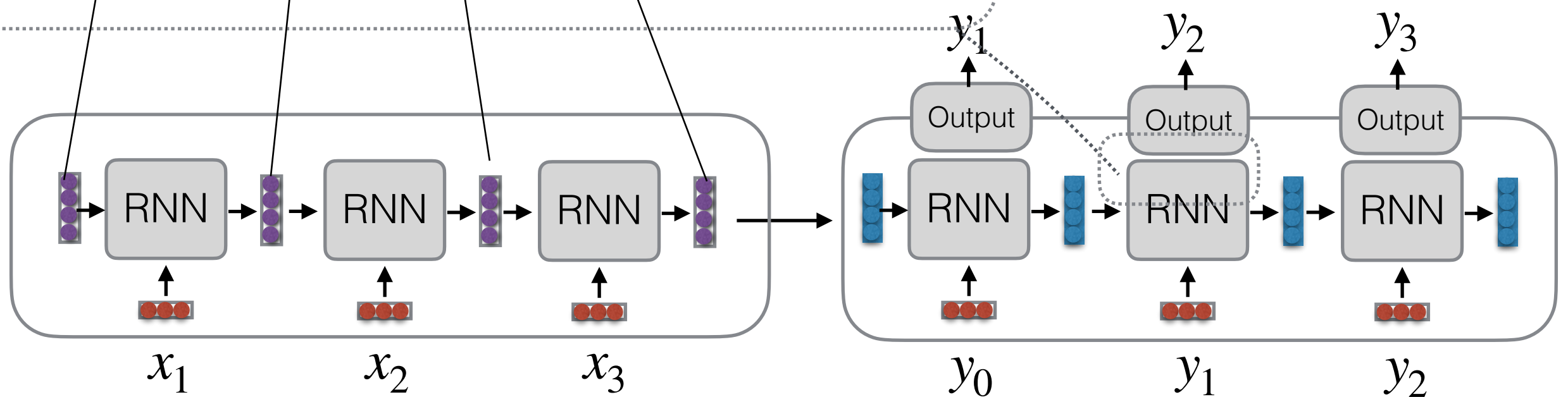
Compute  
attention scores

Context vector is a  
weighted sum



Example usage:

$$\text{logits} = \tanh(W_{\text{out}}[c_t; h_t])$$



Encoder

Decoder

# A Graphical Example

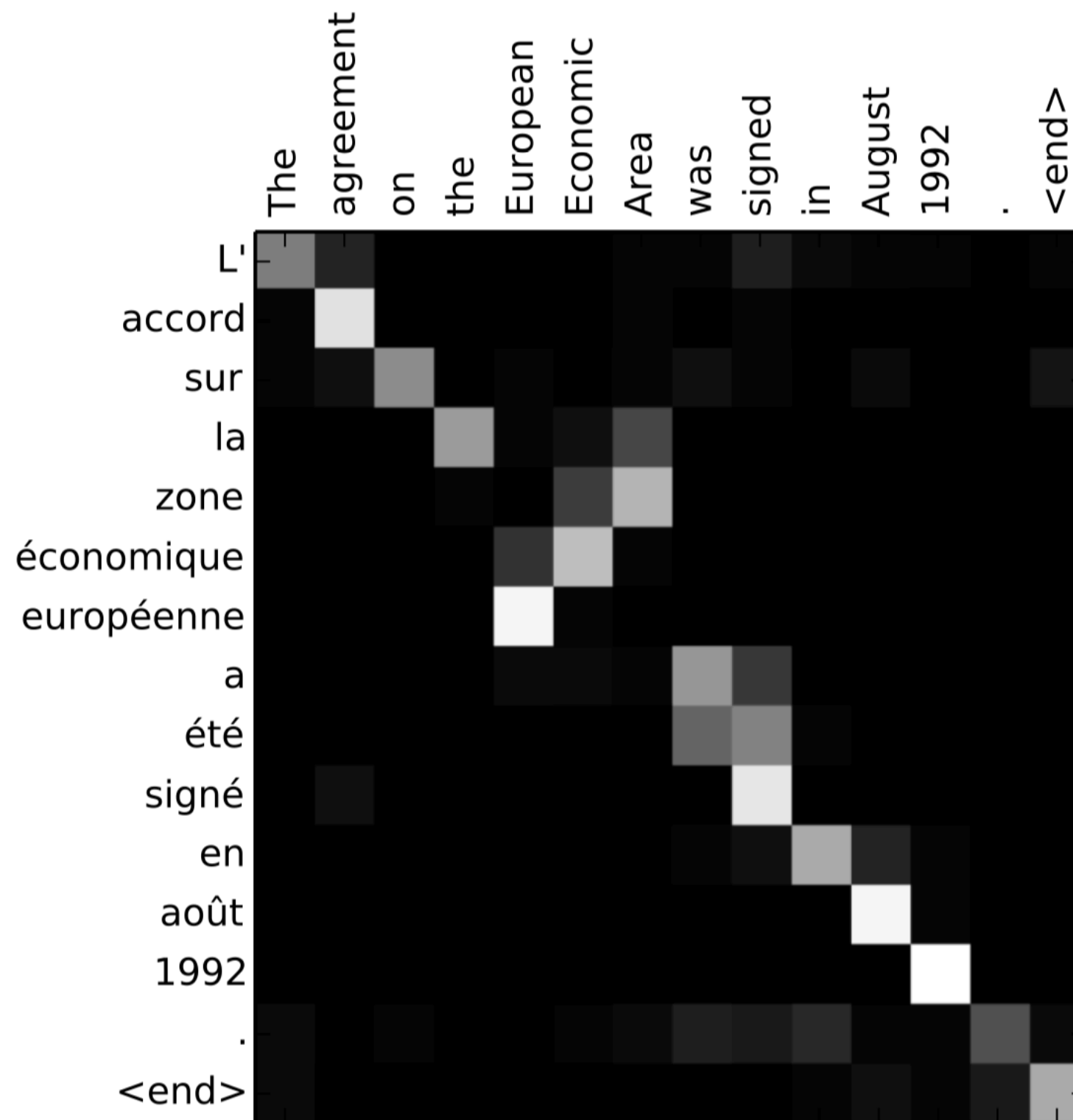


Image from Bahdanau et al. (2015)

# In code

```
class DotAttention(nn.Module):
    def __init__(self):
        super(DotAttention, self).__init__()

    def forward(self, query, keys, values):
        # query: (B, Ty, D)
        # keys: (B, Tx, D)
        # values: (B, Tx, D)
        dot = torch.bmm(keys, query.transpose(1, 2))
        weights = torch.softmax(dot, dim=1)
        out = torch.bmm(weights.transpose(1, 2), values)
        return out, weights
```

# In code

```
def forward(self, X, Yin):  
    # Encode  
    X_embed = self.embed(X)  
    Henc, henc_last = self.encoder(X_embed)  
  
    # Decode  
    Yin_embed = self.embed(Yin)  
    Hdec, _ = self.decoder(Yin_embed, henc_last)  
  
    # Attention  
    query = self.query(Hdec)  
    context, _ = self.attention(query, Henc, Henc)  
  
    # Combine  
    out = torch.cat([Hdec, context], dim=2)  
    out = self.out(out)  
    return out
```

## Attention Visualization

### String Reversal (noise tokens in red)

```
rnommuudloutsv ->
    sulumor
```

# Recap

- Basic encoder-decoder: encode a sequence into a context vector, use it in the decoder
- Attention: context vector is a weighted sum of vectors
  - Using the hidden state as the “query” vector lets us compute a new context vector at each step
- Attention is a general idea: e.g., next lecture we’ll see other variants and uses



# Recap

- Recurrent neural networks
- Vanishing gradients and other recurrent architectures
- Encoder-decoder
- Attention

Time permitting: extra topics

# Overfitting

- Goal: fit a target distribution  $p_*$ 
  - The model may fit the training data (a sample from  $p_*$ ), but the model may not generalize
- **Symptom:** training loss is decreasing, validation loss is increasing
  - Choose different hyperparameters
  - Add regularization
  - Choose the model with minimum validation loss

# Initialization

- Weight initialization impacts the optimization trajectory

```
class DeepCBoW(torch.nn.Module):
    def __init__(self, vocab_size, num_labels, emb_size, hid_size):
        super(DeepCBoW, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.linear1 = nn.Linear(emb_size, hid_size)
        self.output_layer = nn.Linear(hid_size, num_labels)

        nn.init.xavier_uniform_(self.embedding.weight)
        nn.init.xavier_uniform_(self.linear1.weight)
        nn.init.xavier_uniform_(self.output_layer.weight)

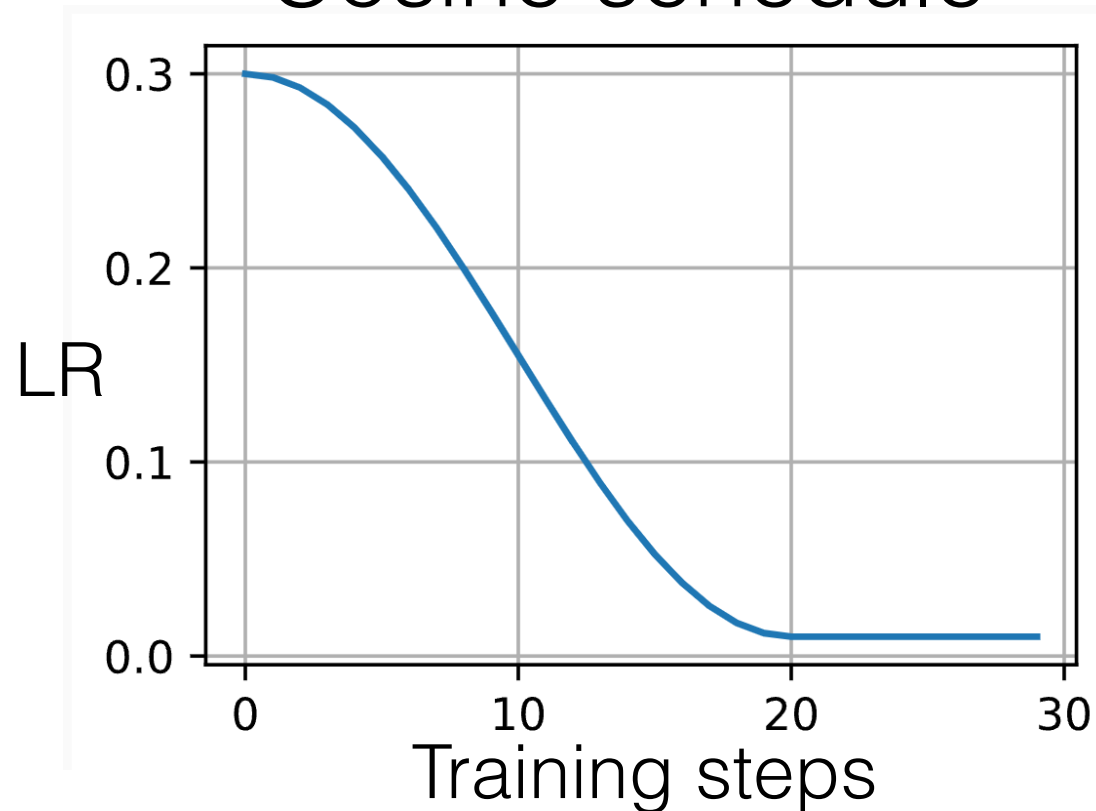
    def forward(self, tokens):
        emb = self.embedding(tokens)
        emb_sum = torch.sum(emb, dim=0)
        h = emb_sum.view(1, -1)
        h = torch.tanh(self.linear1(h))
        out = self.output_layer(h)
        return out
```

Xavier initialization [Glorot and Bengio 2010]:  $W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$

Weights are drawn from a uniform distribution around zero, scaled to balance variance across layers.

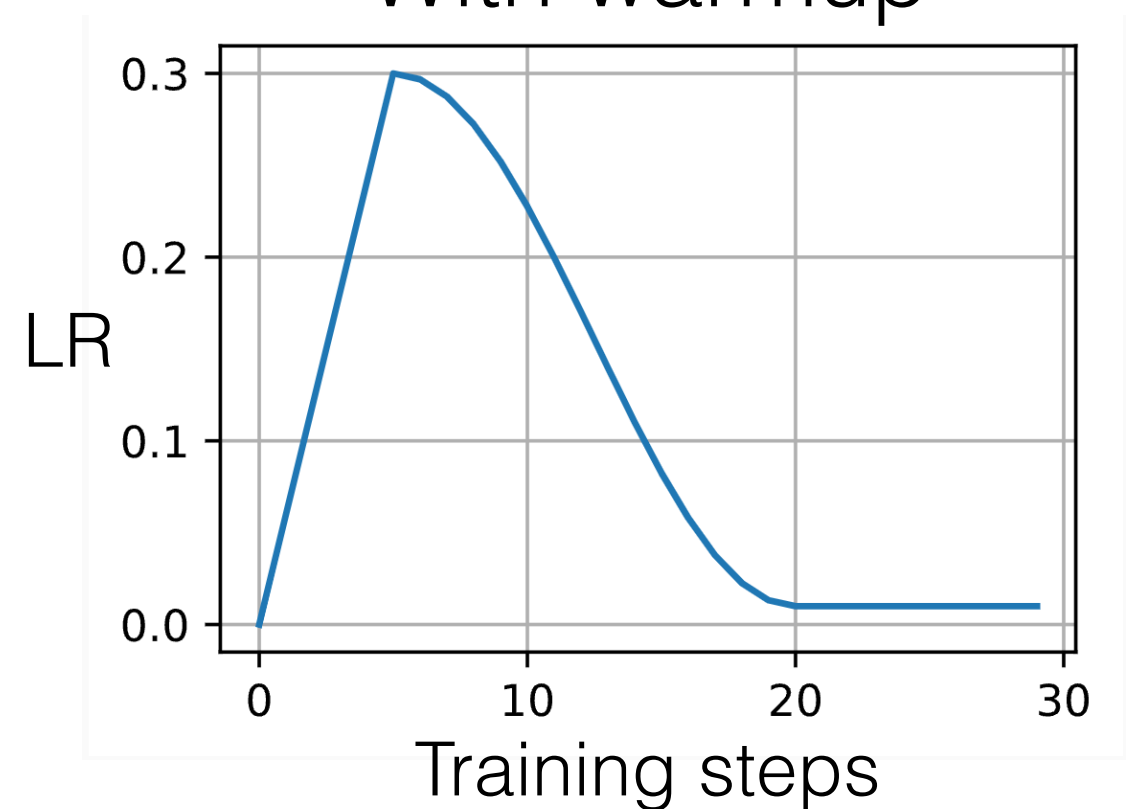
# Learning rate schedule & warmup

## Cosine schedule



- A *schedule* can help balance between exploration (large updates) and convergence (small updates)

## With warmup



- *Warmup* can help stabilize gradients early in training

# Batching

- We typically process multiple examples at once (a batch)
- Takes advantage of parallel hardware (GPU)
- Can smooth out noise in individual gradients

example 1

example 2

example 3

...

example B

```
x_batch = X_train[:8]  
x_batch
```

✓ 0.0s

```
tensor([[26, 26, 26, 26, 26],  
        [26, 26, 26, 26, 11],  
        [26, 26, 26, 11, 20],  
        [26, 26, 11, 20, 0],  
        [26, 11, 20, 0, 13],  
        [11, 20, 0, 13, 13],  
        [26, 26, 26, 26, 26],  
        [26, 26, 26, 26, 18]])
```

# Batching

- When *inputs* are of variable length, we use a *pad token*

```
tensor([[26, 11, 20,  0, 13, 13, 27, 27, 27, 27],
        [26, 18,  7,  0,  8, 13, 27, 27, 27, 27],
        [26, 17, 20, 15,  4, 17, 19, 27, 27, 27],
        [26, 12, 14, 10, 18,  7,  0,  6, 13,  0]])
['[S]', 'l', 'u', 'a', 'n', 'n', '[PAD]', '[PAD]', '[PAD]', '[PAD]']
['[S]', 's', 'h', 'a', 'i', 'n', '[PAD]', '[PAD]', '[PAD]', '[PAD]']
['[S]', 'r', 'u', 'p', 'e', 'r', 't', '[PAD]', '[PAD]', '[PAD]']
['[S]', 'm', 'o', 'k', 's', 'h', 'a', 'g', 'n', 'a']
```

- We may need to *mask out* operations involving pad tokens

```
def forward(self, words, mask):
    emb = self.embedding(words)
    # Mask out the padding tokens
    emb = emb * mask.unsqueeze(-1)
    h = torch.sum(emb, dim=1)
    for i in range(self.nlayers):
        h = torch.relu(self.linears[i](h))
        h = self.dropout(h)
    out = self.output_layer(h)
    return out
```

# Batching

- When *outputs* are of variable length, we mask out the loss for pad tokens

```
# NOTE: We ignore the loss whenever the target token is a padding token  
criterion = nn.CrossEntropyLoss(ignore_index=token_to_index['[PAD]'])
```

We'll see a concrete example next class!

Thank you!