CS11-711 Advanced NLP

Language and Sequence Modeling II

Sean Welleck

Carnegie Mellon University
Language Technologies Institute

https://cmu-l3.github.io/anlp-spring2025/

Some slides adapted from Graham Neubig from Fall 2024

Recap

- Language modeling
  - Model a distribution of sequences (e.g., text)
  - N-gram models and feedforward model
    - 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

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

Next Token

Full context

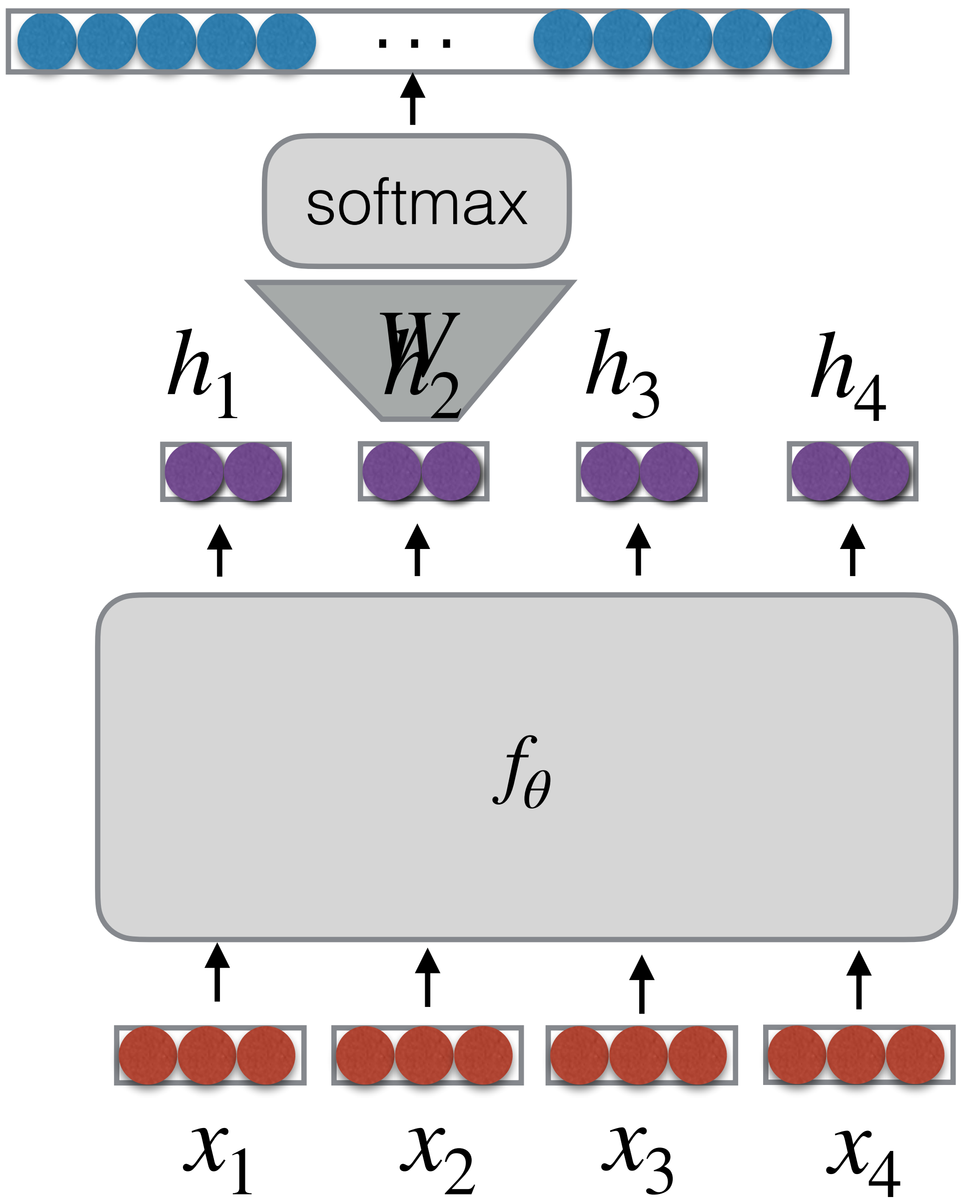
Outline

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

Recurrent Neural Networks

Sequence Model

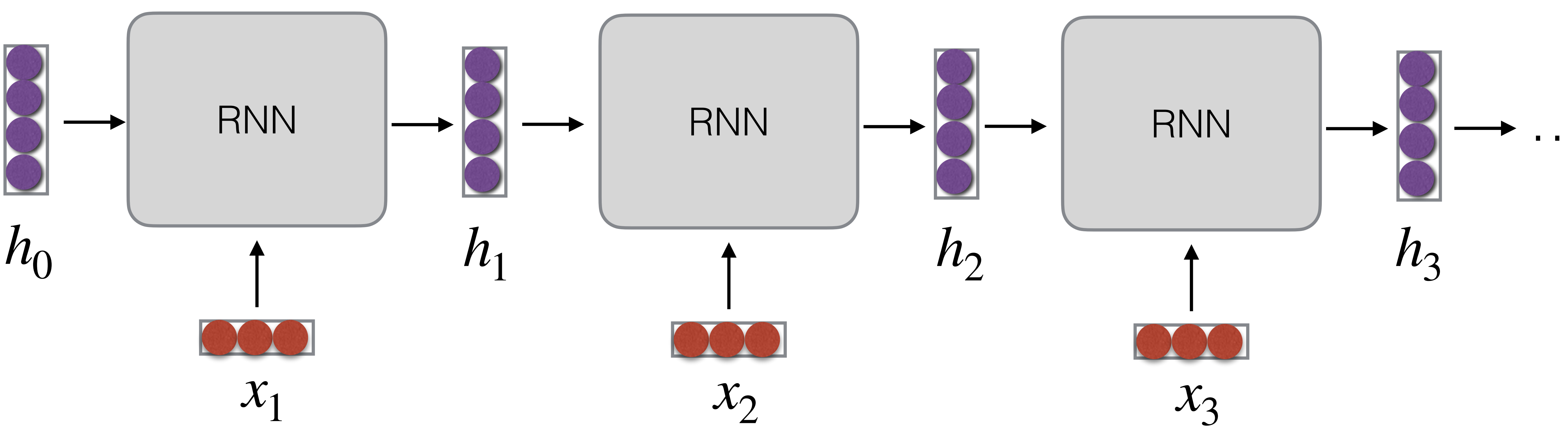
- $f_\theta(x_1, \ldots, x_{|x|}) \rightarrow h_1, \ldots, h_{|x|}$
  - $h_t \in \mathbb{R}^d$: hidden state
- Language modeling:
  - $p_\theta(\cdot \mid x_{<t}) = \text{softmax}\left(W h_t^\top\right)$



Recurrent neural network

$$h_t = \sigma\left(W_h h_{t-1} + W_x x_t + b\right)$$

σ : activation function (tanh, relu, …)



Parameters θ $W_h \in \mathbb{R}^{d \times d}$

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

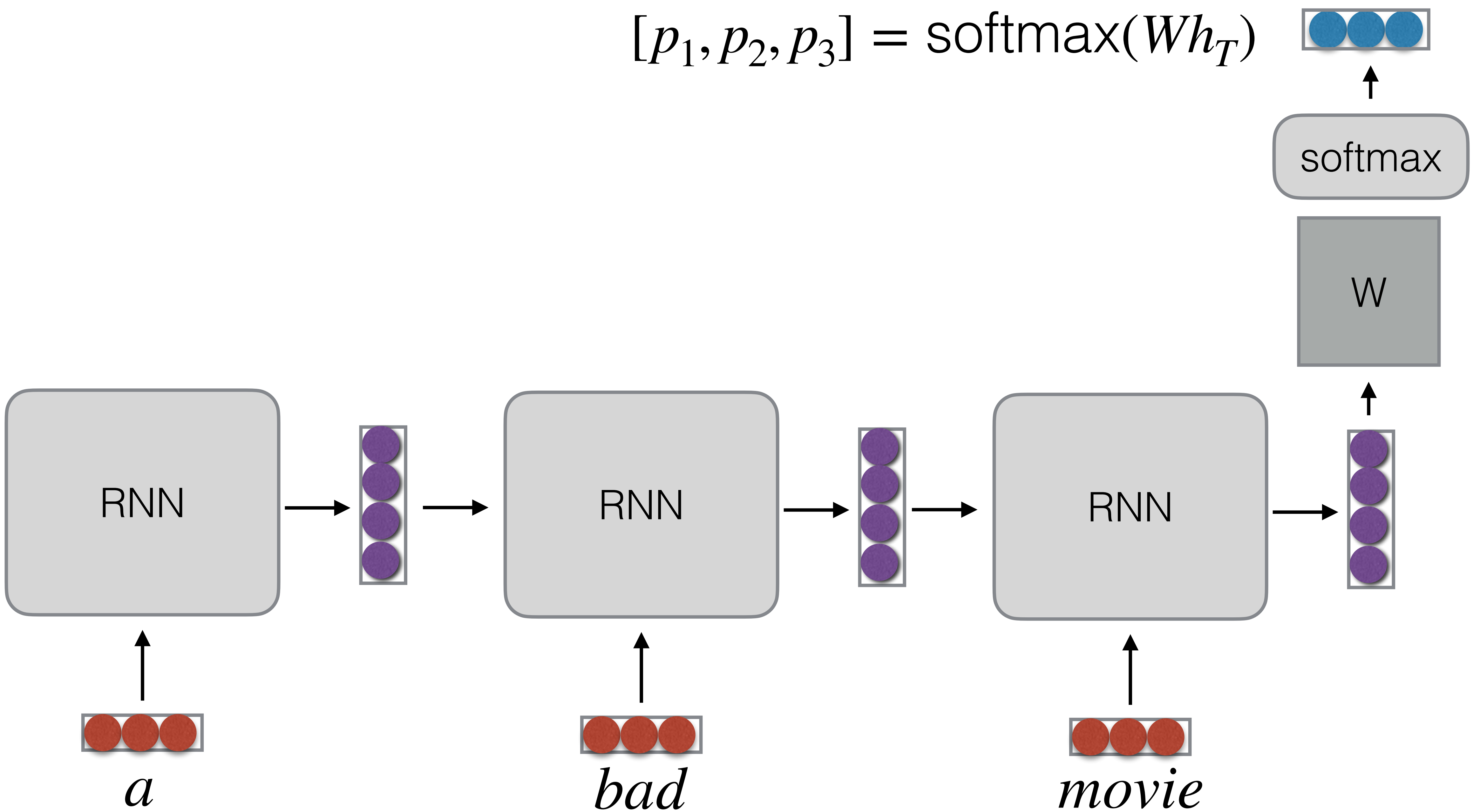
$b \in \mathbb{R}^{d}$

Elman 1980

Example: sequence classification

Output class probabilities

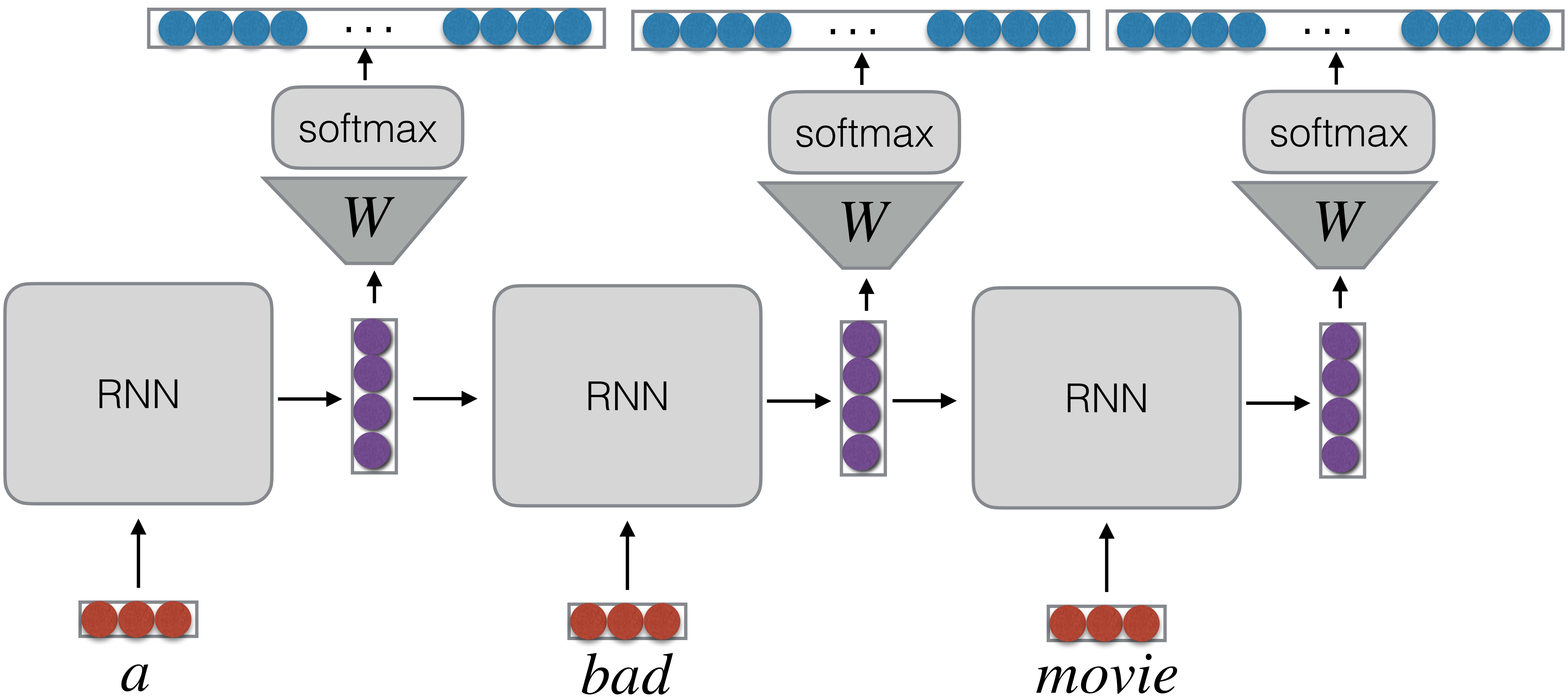
$[p_1, p_2, p_3] = \text{softmax}(Wh_T)$



Example: language modeling

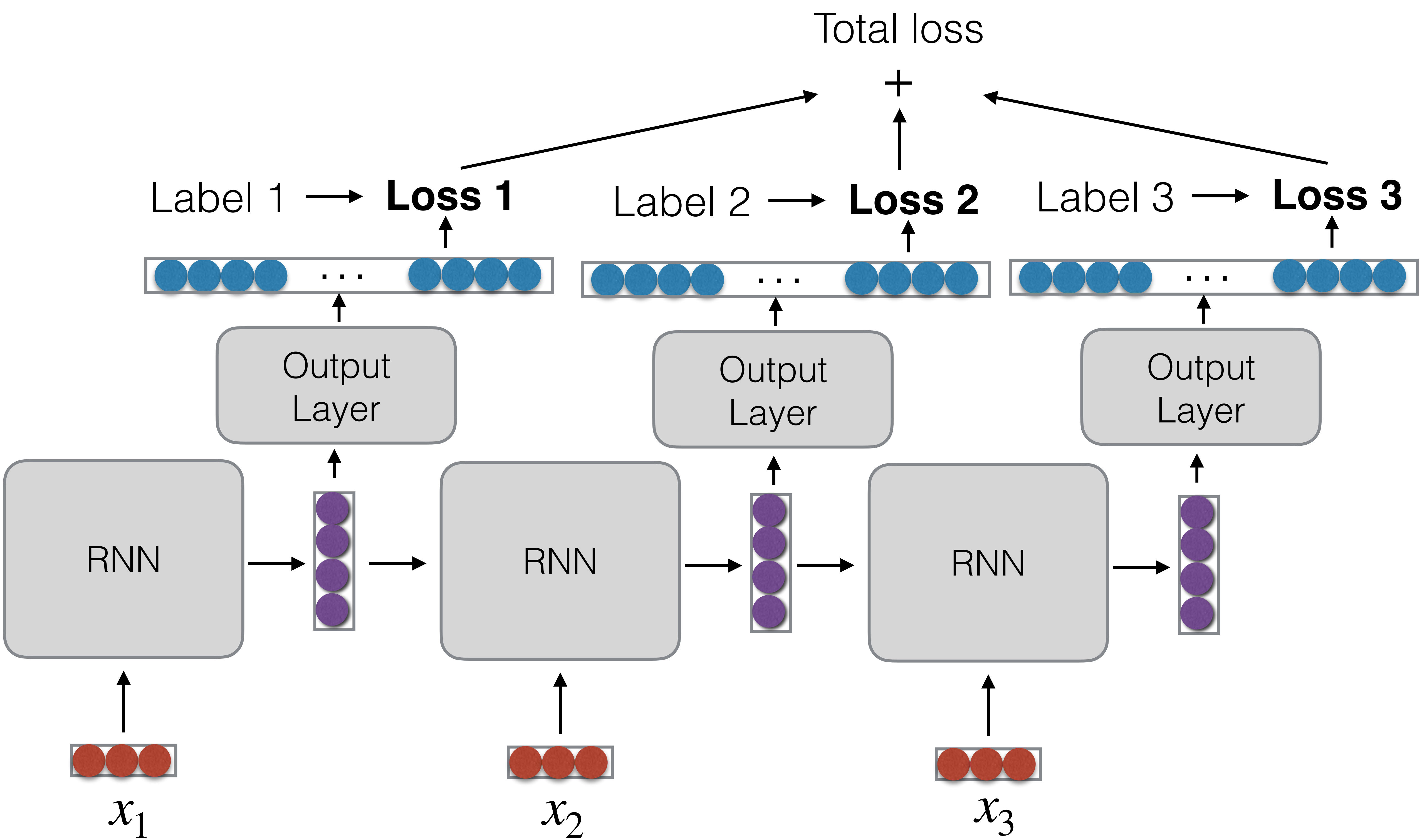
Next-token probabilities

$p(x_t | x_{<t}) = \text{softmax}(Wh_t)$



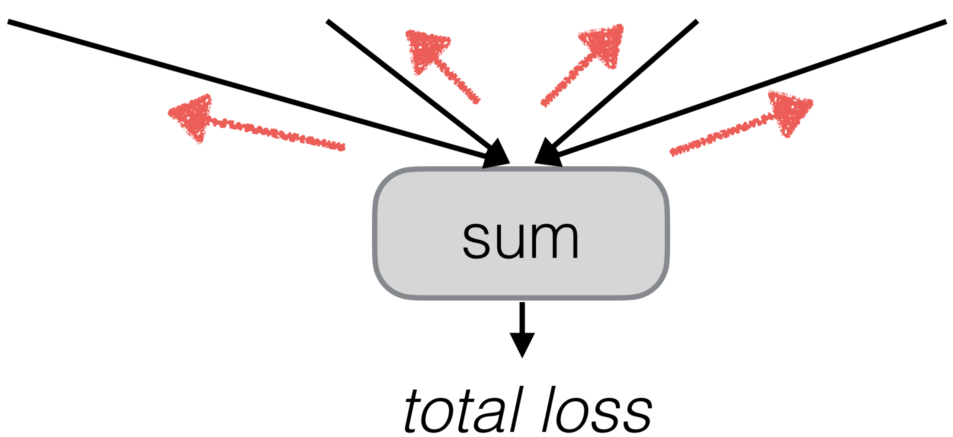
Mikolov et al 2010, Recurrent neural network based language model

Training RNNs



RNN Training

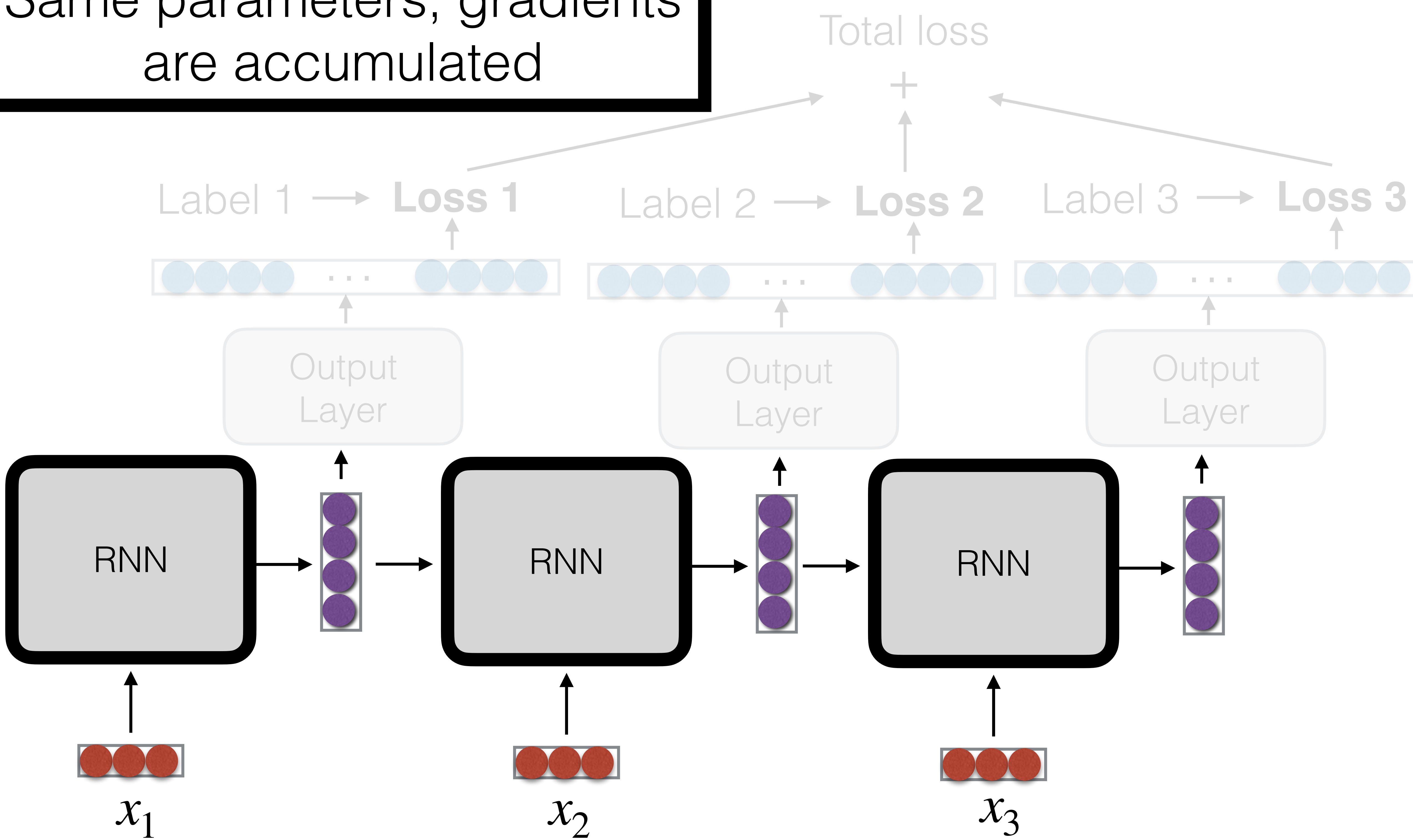
- The unrolled graph is a well-formed (DAG) computation graph—we can run backpropagation



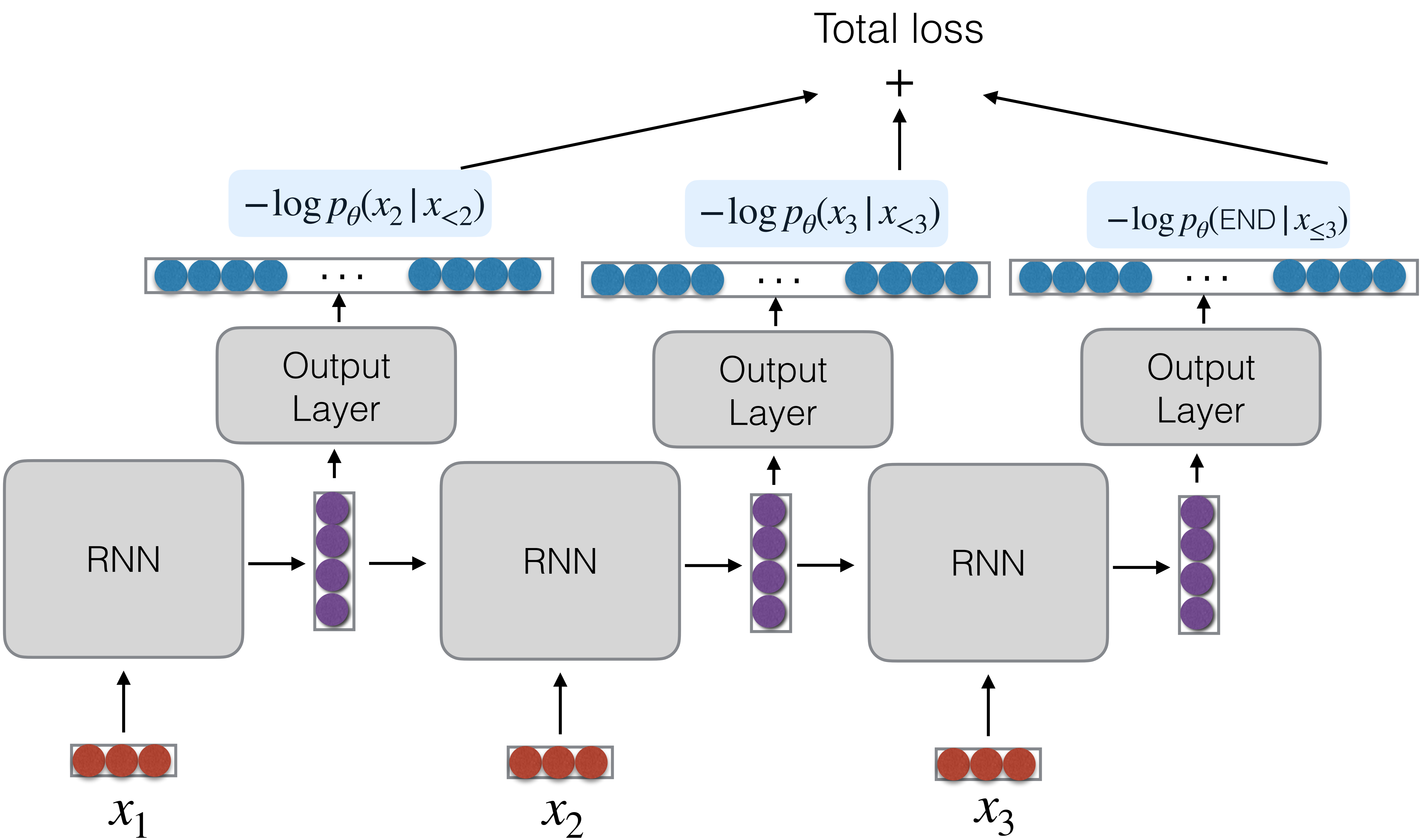
- This is historically called “backpropagation through time” (BPTT)

Parameter tying

Same parameters; gradients are accumulated



Training RNNs: Language Modeling



Training RNNs: Language Modeling

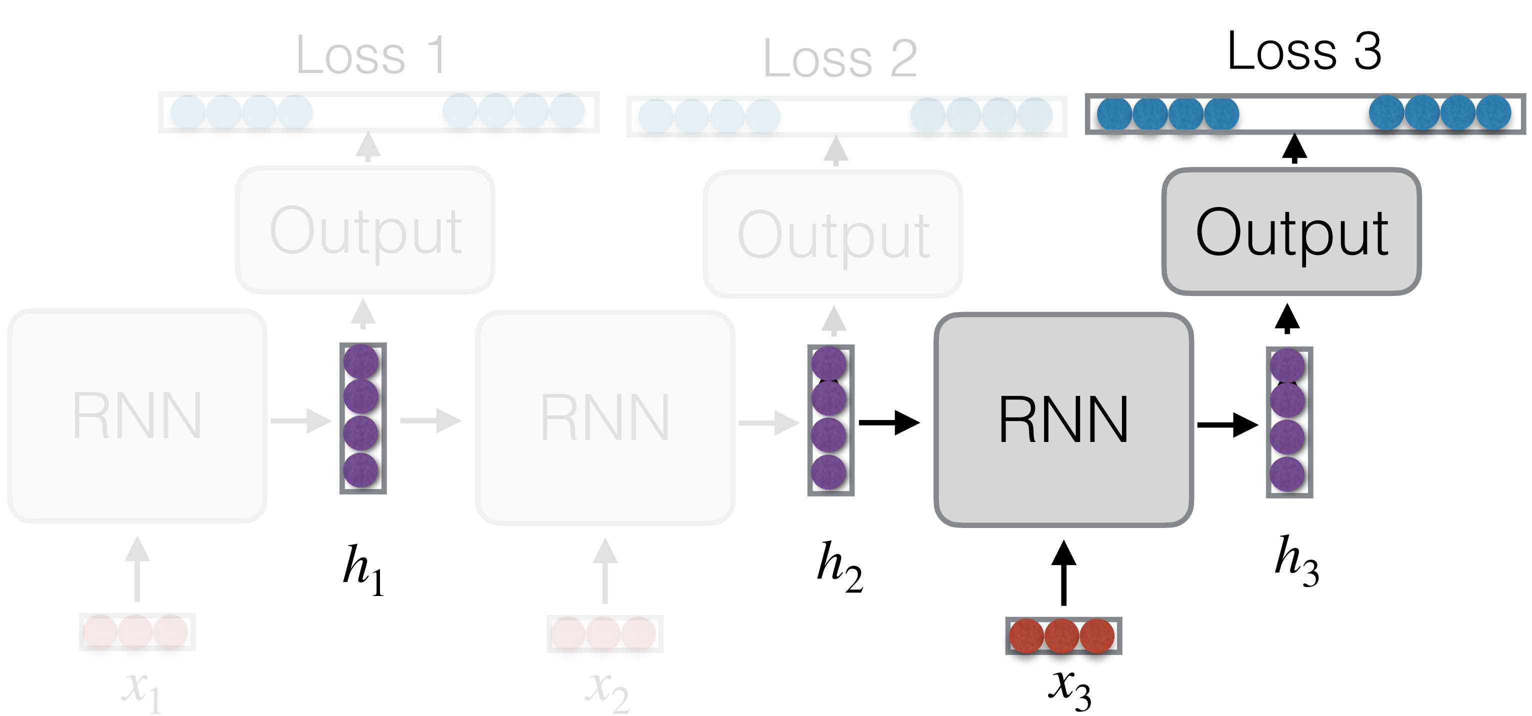
- Maximum likelihood estimation (again!)
- $$\max \sum_{x \in D_{train}} \log p_\theta(x)$$

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

Previous slide

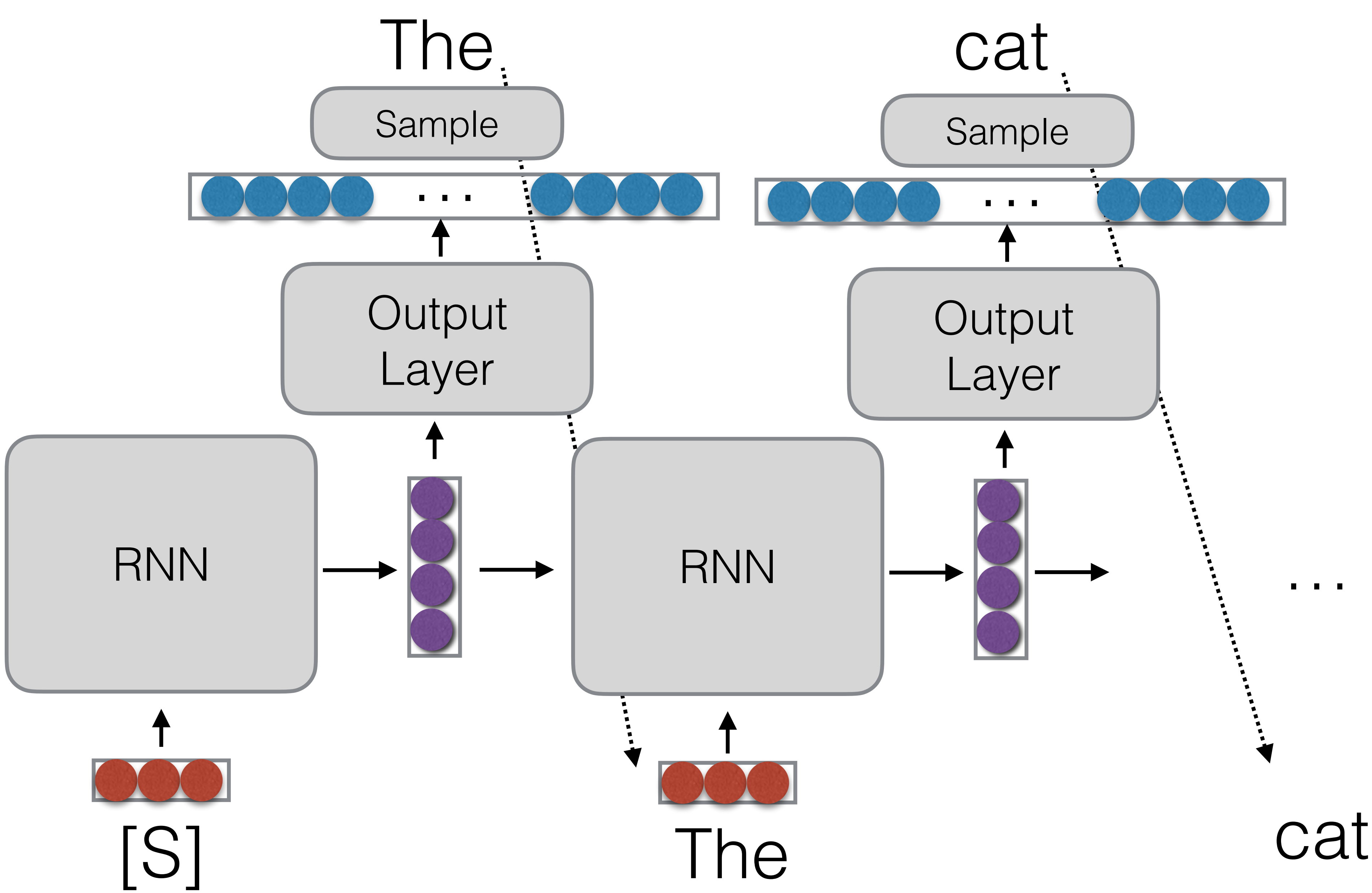
Training RNNs: Language Modeling

- 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

We’ll compare this to transformers in the next lecture!

Recap: RNNs

- A sequence model, $f_\theta(x_1, \ldots, x_{|x|}) \rightarrow h_1, \ldots, h_{|x|}$
- Transforms a *hidden state* at each step
  - $h_t = \sigma\left(W_h h_{t-1} + W_x x_t + b\right)$
  - 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

```python
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
```

https://github.com/cmu-l3/anlp-spring2025-code/blob/main/04_recurrent/recurrent_lm.ipynb

In Code

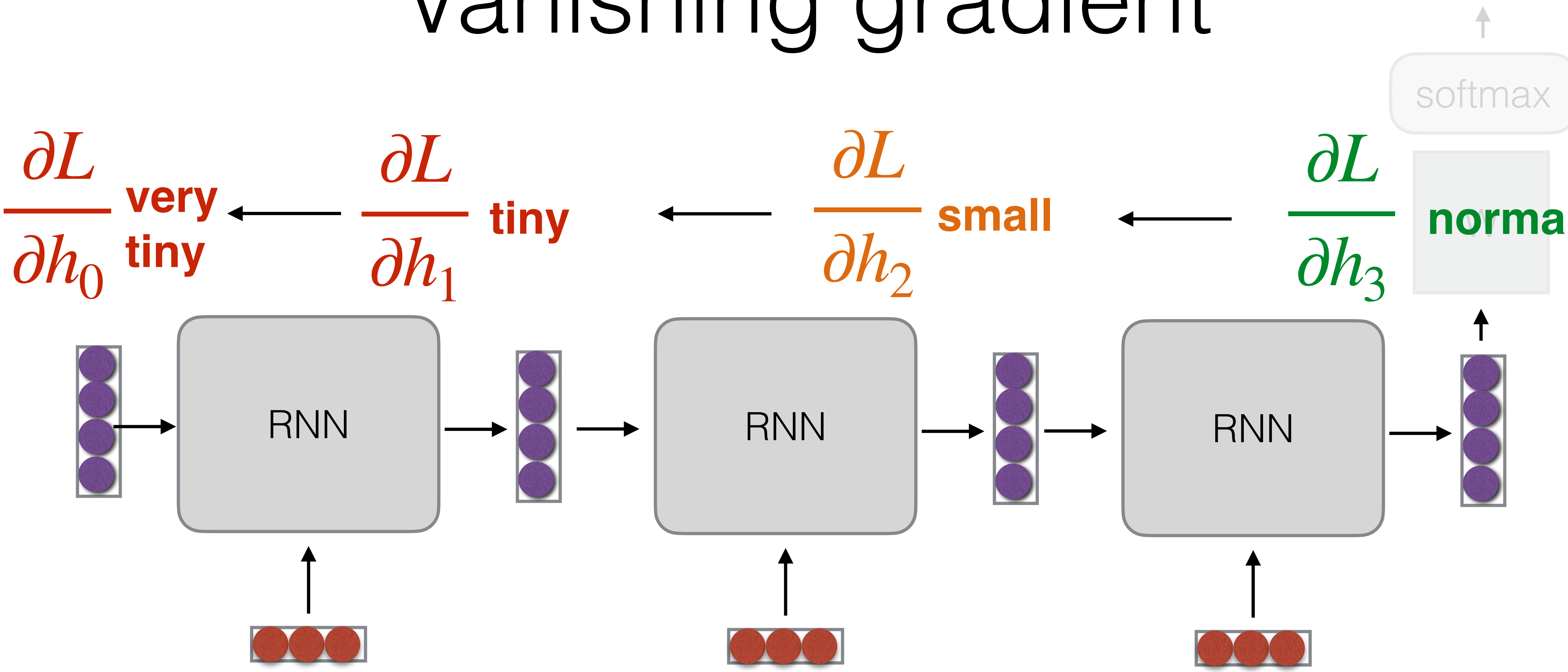
```python
class RNNLM(nn.Module):
    def __init__(self, vocab_size, hidden_size):
        super(RNNLM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.rnn = RNNCell(hidden_size, hidden_size)
        self.output = nn.Linear(hidden_size, vocab_size)
        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: $h_t = \tanh(W_{in}x + Wh_t)$, $y_T = W_{out}h_T$

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

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

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

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

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

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 $z \to 0$ for multiple steps:
  - $$\frac{\partial h_{t_2}}{\partial h_{t_1}} = \prod_{t=t_1}^{t_2} \underbrace{\frac{\partial h_t}{\partial h_{t-1}} h_{t-1}}_{1} = 1$$

A solution: gating and additive connections

- **Basic idea:** pass information across timesteps with a learned "gate" z_t

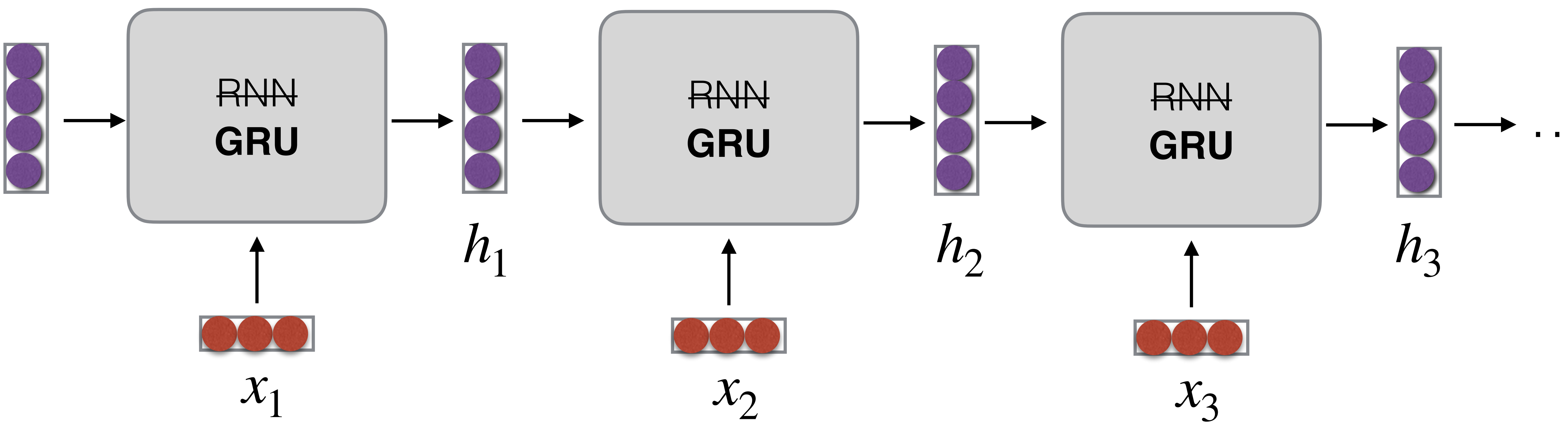
    - $h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t$

- 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"

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

- "Reset gate"

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

- Recurrent update:

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

- $\hat{h}_t$ is a "candidate state"

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

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

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

```python
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

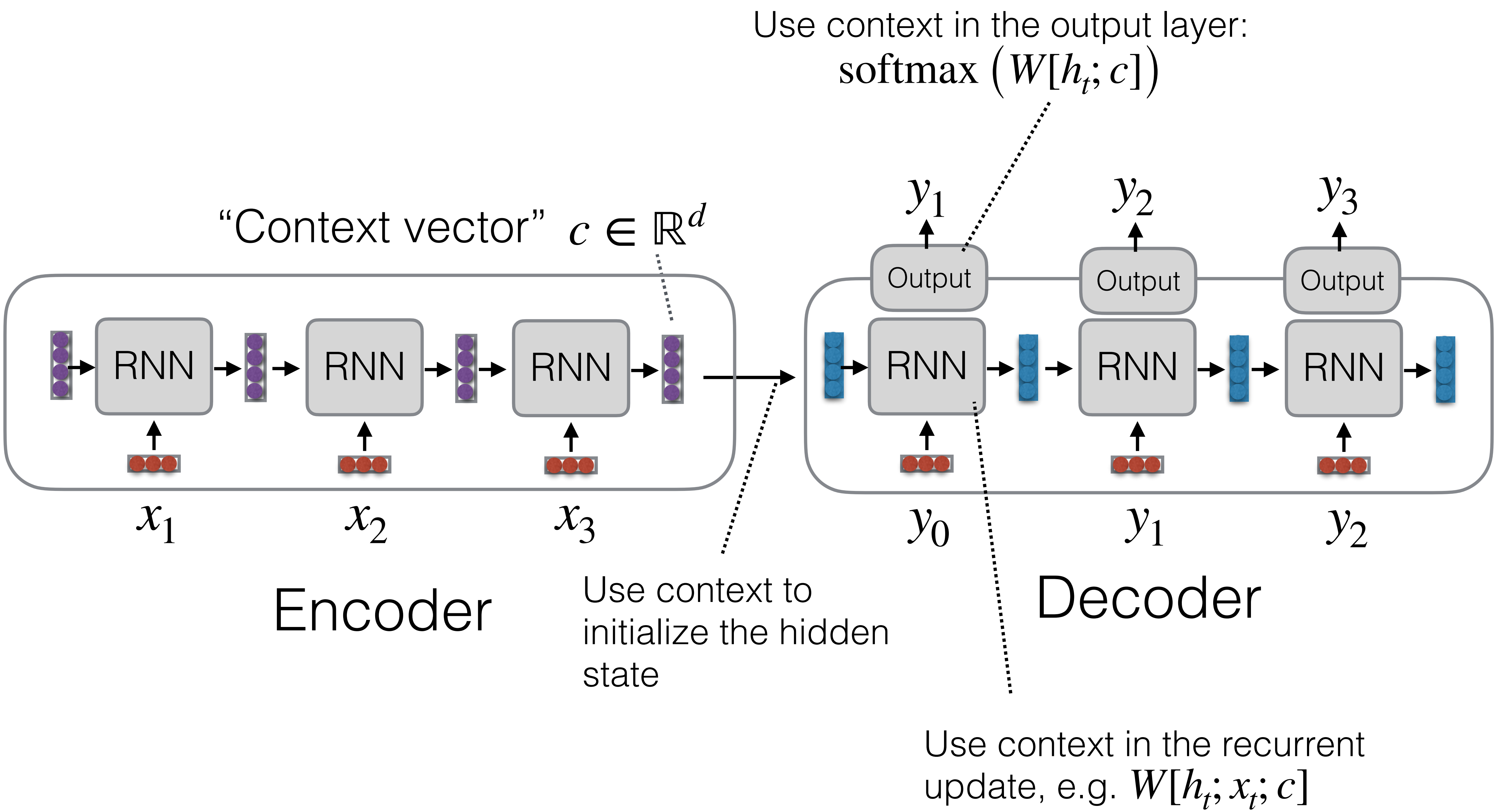
- Motivation: conditional generation

$$p_\theta(y_1, \ldots, y_T \mid x)$$

$y_1, \ldots, y_T$	x
Japanese sentence	English sentence
Response	Chat history
…	…

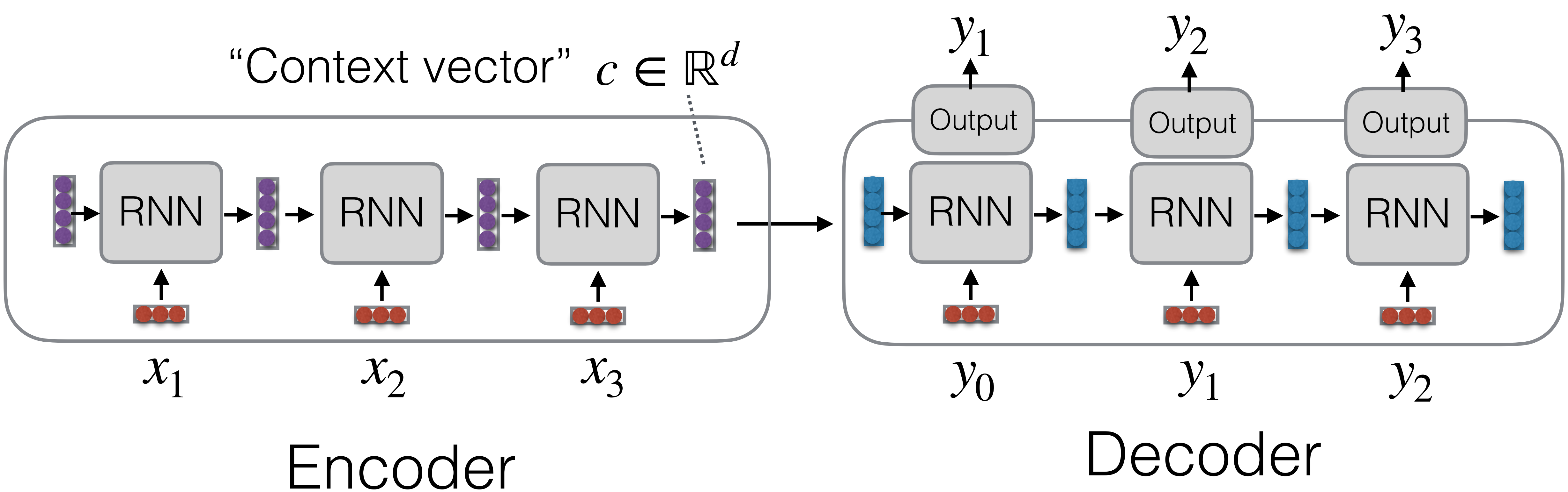
- Basic idea: use a sequence model to represent x as a vector

Encoder-decoder



Cho et al 2014, Learning Phrase Representations using RNN Encoder–Decoder

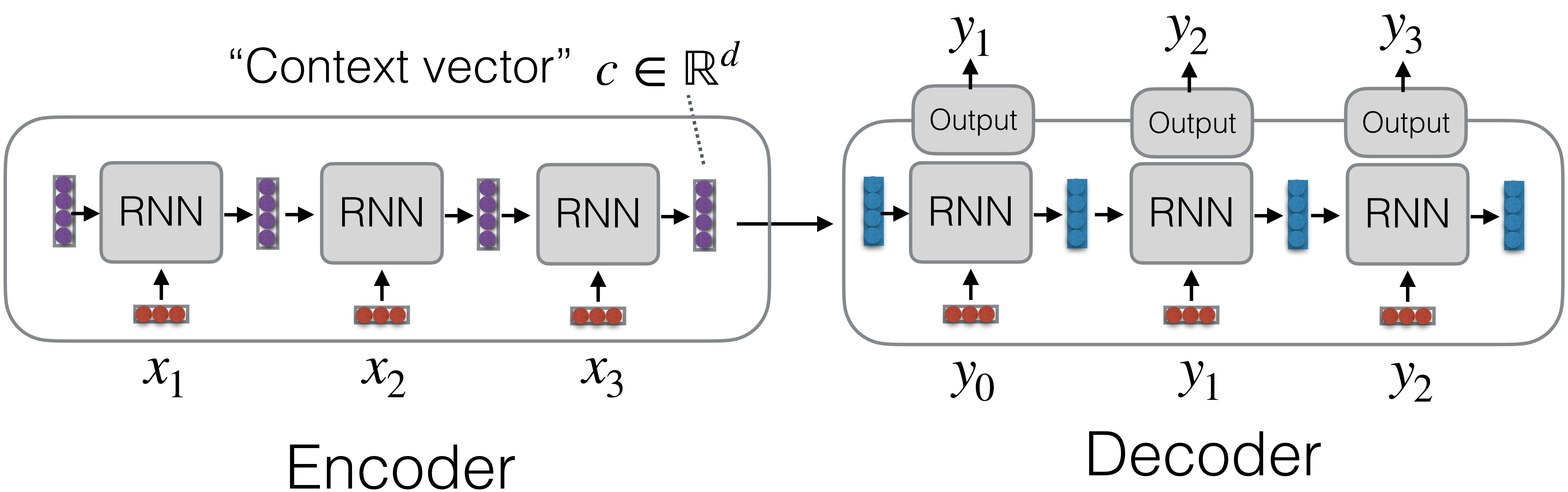
Encoder-decoder



Training:

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

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”

Bahdanau et al 2015, Neural Machine Translation by Jointly Learning to Align and Translate

Attention

- **Keys**: Encoder states $h_1^{enc}, \ldots, h_N^{enc}$
- **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

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

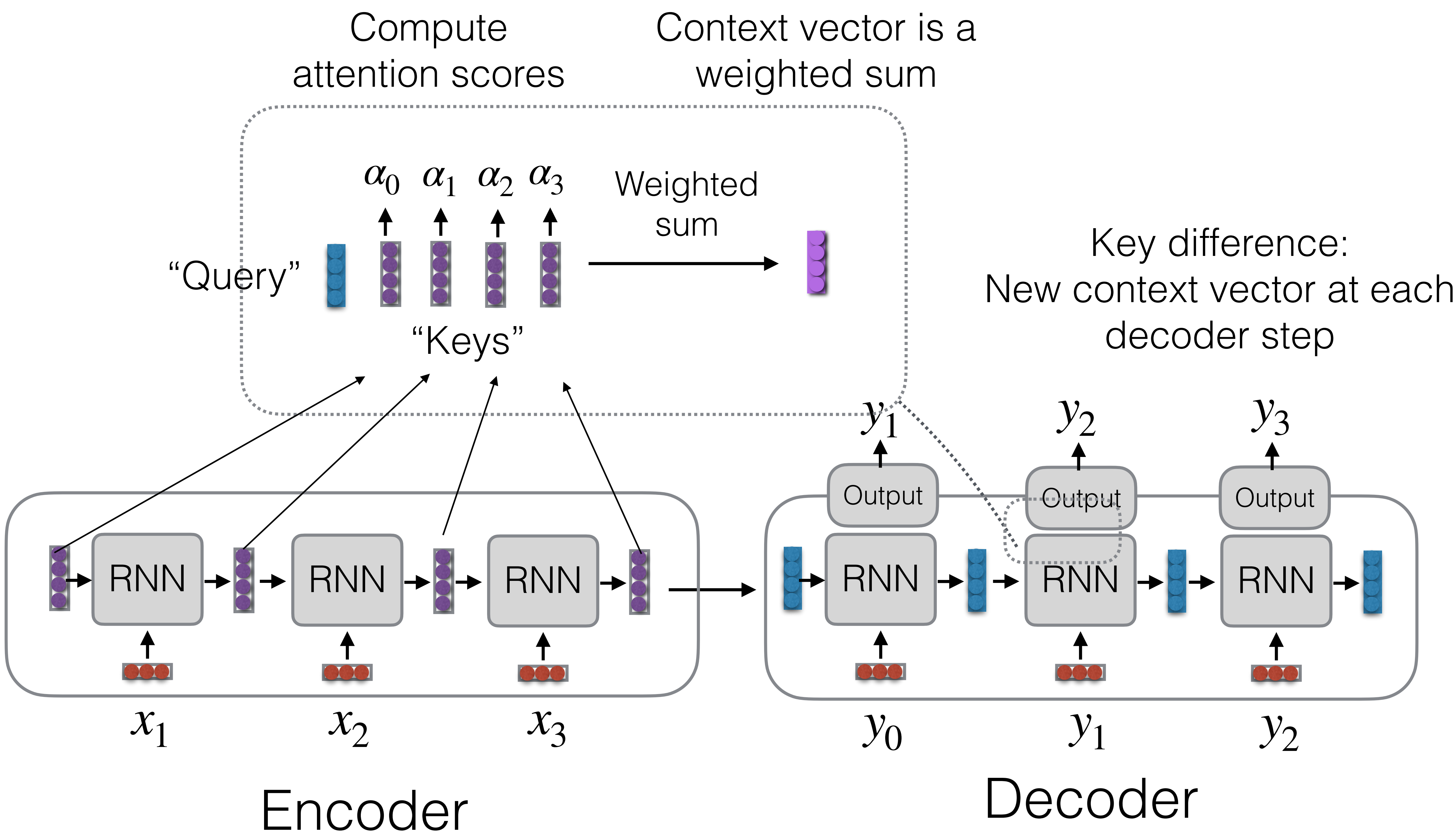
Bilinear

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

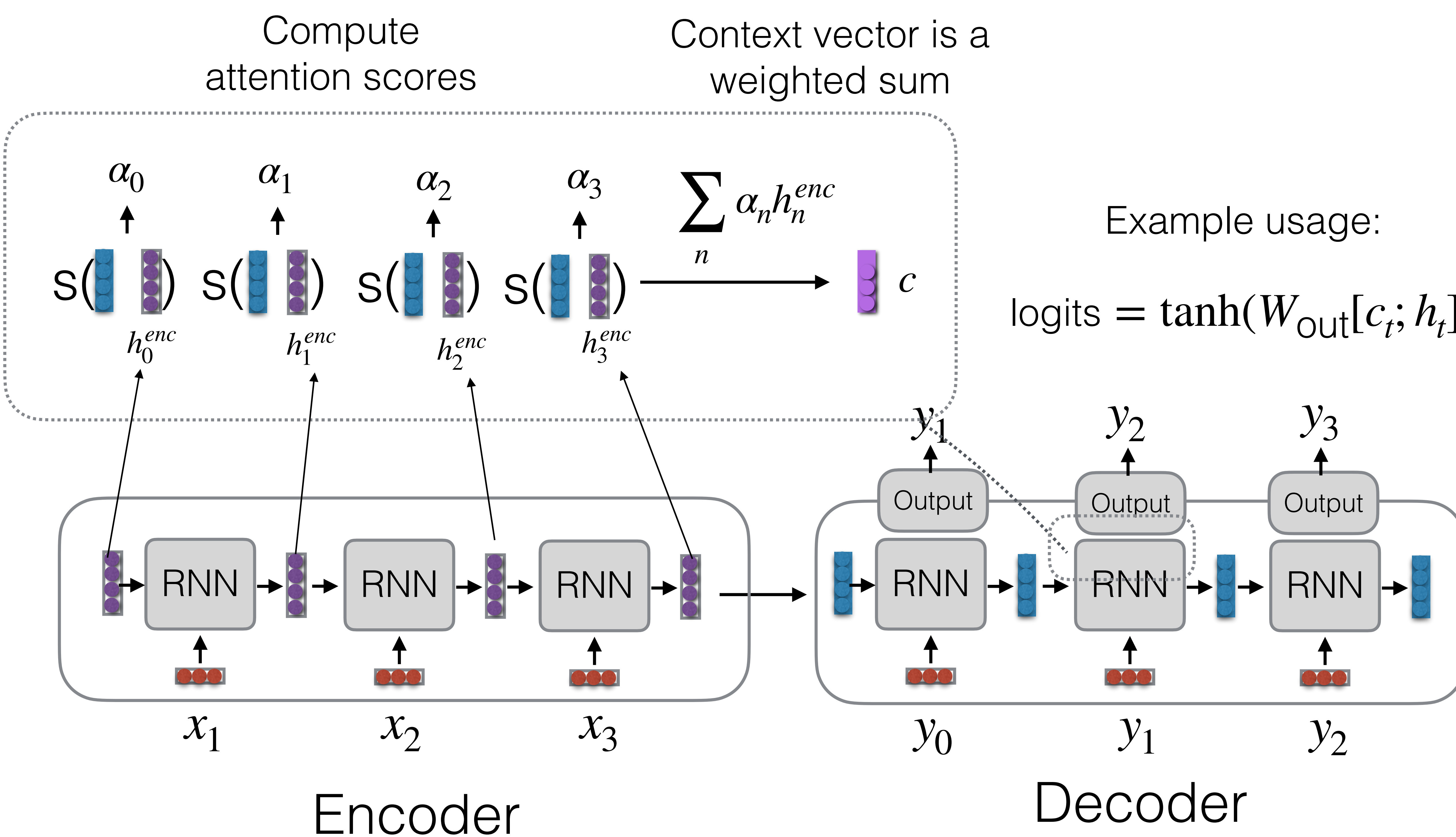
Nonlinear

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

Attention


Compute
attention scores
Context vector is a
weighted sum
α_0 α_1 α_2 α_3
Weighted
sum
"Query"
"Keys"
Key difference:
New context vector at each
decoder step
y_1 y_2 y_3
Output
Output
Output
RNN
RNN
RNN
RNN
RNN
RNN
x_1 x_2 x_3
y_0 y_1 y_2
Encoder
Decoder

Attention



Example usage:

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

A Graphical Example

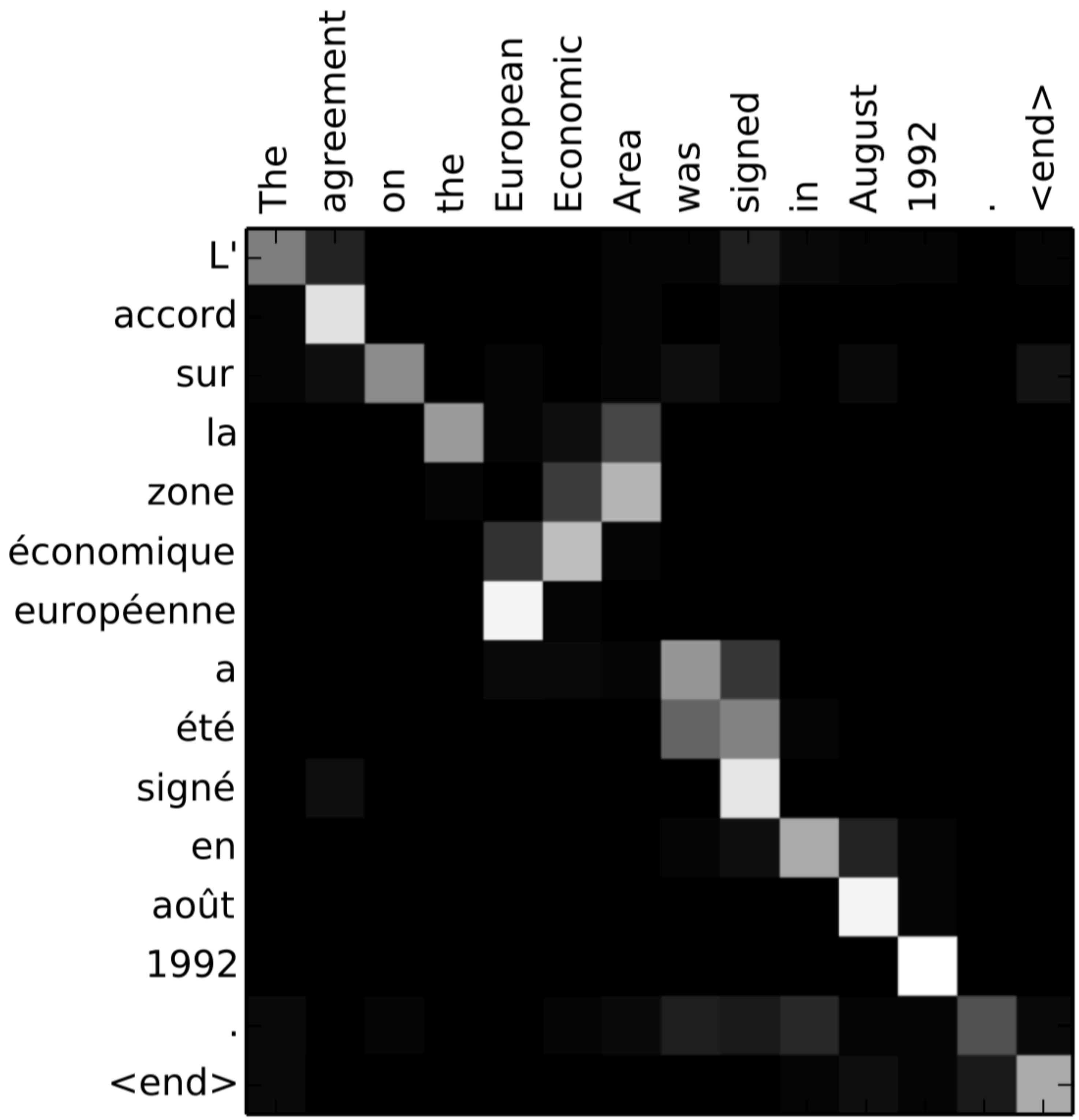


Image from Bahdanau et al. (2015)

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
```

https://github.com/cmu-l3/anlp-spring2025-code/blob/main/04_recurrent/recurrent_encdec.ipynb

In code

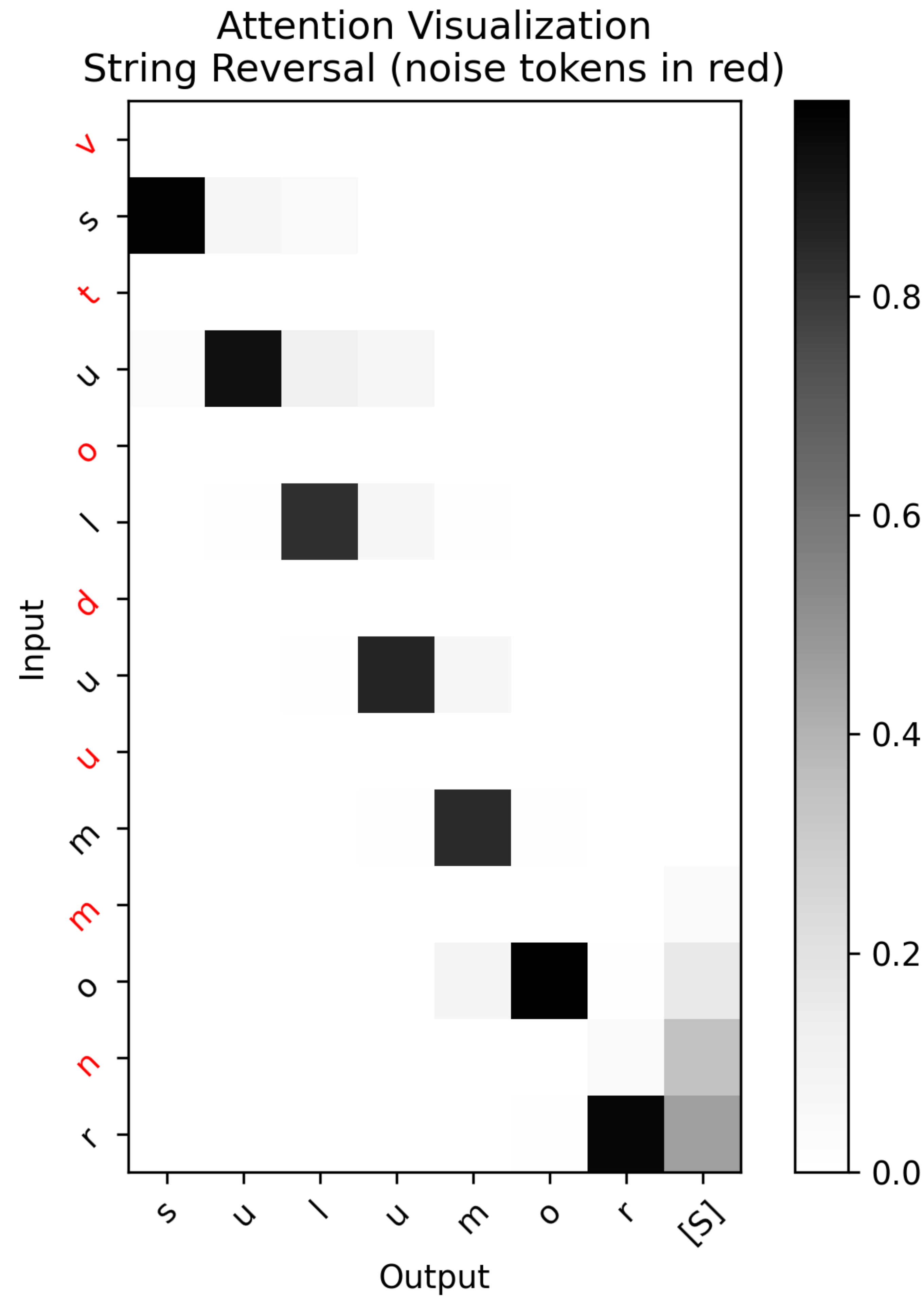
```
def forward(self, X, Yin):
    # Encode
    X_embed = self.embed(X)
    Henc, henc_last = self.encoder(X_embed)

    # 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
```

Task

Reverse a name that has noise characters

romulus -> sulumor

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

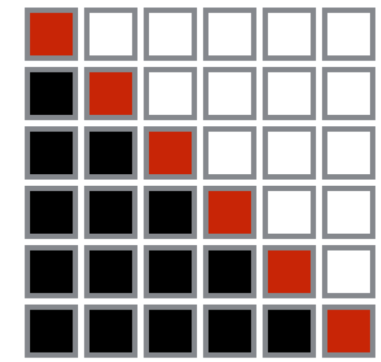
Time permitting: extra topics

Types of Unconditioned Sequence Modeling

Left-to-right Autoregressive Prediction

$$P(X) = \prod_{i=1}^{|X|} P(x_i | x_1, \ldots, x_{i-1})$$

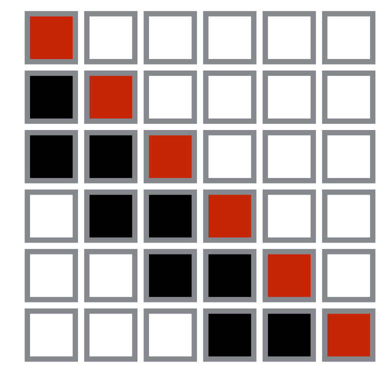
(e.g. RNN or Transformer LM)



Left-to-right Markov Chain (order n-1*)*

$$P(X) = \prod_{i=1}^{|X|} P(x_i | x_{i-n+1}, \ldots, x_{i-1})$$

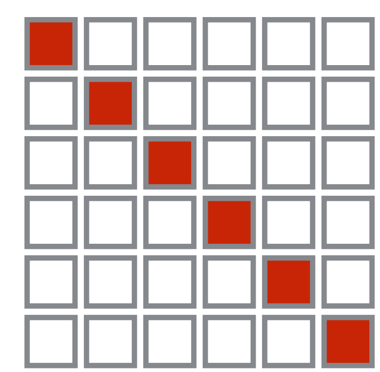
(e.g. n-gram LM, feed-forward LM)



Independent Prediction

$$P(X) = \prod_{i=1}^{|X|} P(x_i)$$

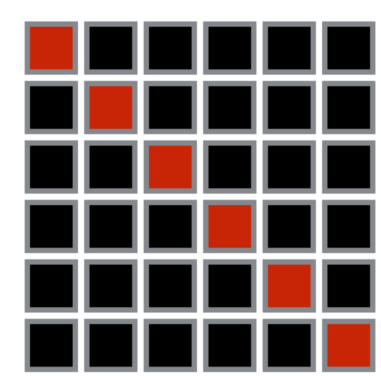
(e.g. unigram model)



Bidirectional Prediction

$$P(X) \neq \prod_{i=1}^{|X|} P(x_i | x_{\neq i})$$

(e.g. masked language model)

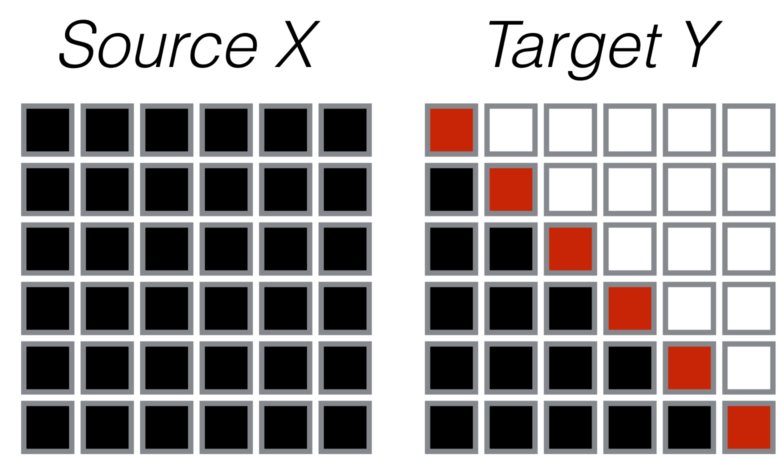


Types of Conditioned Sequence Modeling

Autoregressive

$$P(Y|X) = \prod_{i=1}^{|Y|} P(y_i|X, y_1, \ldots, y_{i-1})$$

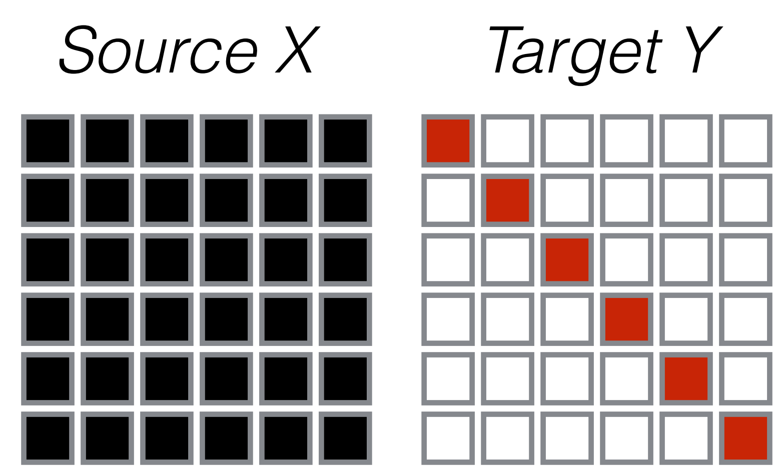
(e.g. seq2seq model)



Non-autoregressive

$$P(Y|X) = \prod_{i=1}^{|Y|} P(y_i|X)$$

(e.g. sequence labeling, non-autoregressive MT)



Questions?