Deep Neural Networks for Sequential Data

CS771: Introduction to Machine Learning

Piyush Rai

Plan Today

- Batch-normalization and Dropout layers
- Deep Neural Networks for sequential data (e.g., text)
  - Recurrent Neural Networks and variants
  - CNNs for text
  - Attention mechanism

Recap: Normalization Layer

- Each hidden layer is a nonlinear transformation of the previous layer's inputs
- To prevent distribution drift in activations' distribution, we often "standardize" each layer
- Standardize = activation $h_{nk}^{(\ell)}$ should have zero mean and unit variance across all $\boldsymbol{n}$
- It is achieved by inserting a "batch normalization" layer after each hidden layer
- To do so, during training, (omitting layer number ℓ) we replace each $\boldsymbol{h}_n$ by $\widetilde{\boldsymbol{h}}_n$

$\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are trainable batch-norm parameters

$$\widetilde{\boldsymbol{h}}_n = \boldsymbol{\gamma} \odot \widehat{\boldsymbol{h}}_n + \boldsymbol{\beta} \qquad \widehat{\boldsymbol{h}}_n = \frac{\boldsymbol{h}_n - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \epsilon}}$$

Note: Batch-norm assumes sufficiently large mini-batch $\mathcal{B}$ to work well. There are variants such as "layer normalization" and "instance normalization" that don't require a mini-batch can be computed using a single example

We compute $\boldsymbol{\mu}_{\mathcal{B}}$ and $\boldsymbol{\sigma}_{\mathcal{B}}^2$ using the data from the current minibatch of examples $\mathcal{B}$ (thus the name "batch norm"

$$\boldsymbol{\mu}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{h}\in\mathcal{B}} \boldsymbol{h} \qquad \boldsymbol{\sigma}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{h}\in\mathcal{B}} (\boldsymbol{h} - \boldsymbol{\mu}_{\mathcal{B}})^2$$

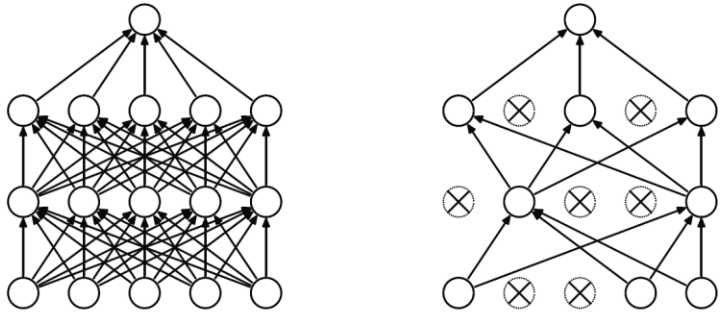
- After training, we store $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ + the statistics $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ computed on the <u>whole training data</u>, and use these values to apply batch-norm on each test input

Dropout Layer

- Deep neural networks can overfit when trained on small datasets
- Dropout is a method to regularize without using an explicit regularizer
- In every update of the network, drop neuron $\boldsymbol{i}$ in layer $\boldsymbol{\ell}$ with probability $\boldsymbol{p}$

$$\epsilon_i^{(\ell)} \sim \text{Bernoulli}(1-p)$$

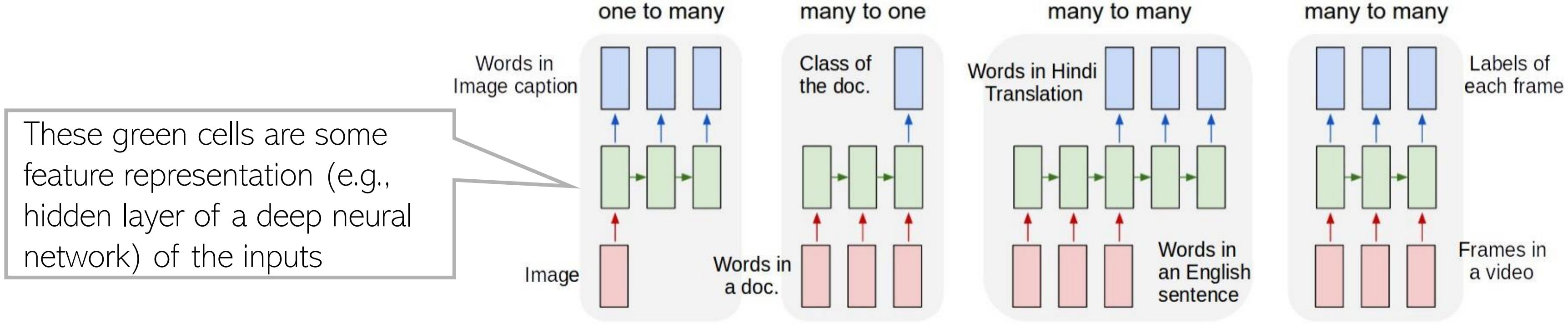
- If $\epsilon_i^{(\ell)} = \mathbf{0}$, set all outgoing weights $w_{ij}^{(\ell)}$ from neuron $\boldsymbol{i}$ to 0



- Each update of weights will change a different subset of weights
  - In doing so, we are making individual neurons more self-reliant and less dependent on others
- At test time, no dropout is used. After training is complete, we multiply each weight by the keep probability $\mathbf{1}-\boldsymbol{p}$ and use these weights for predictions

Sequential Data

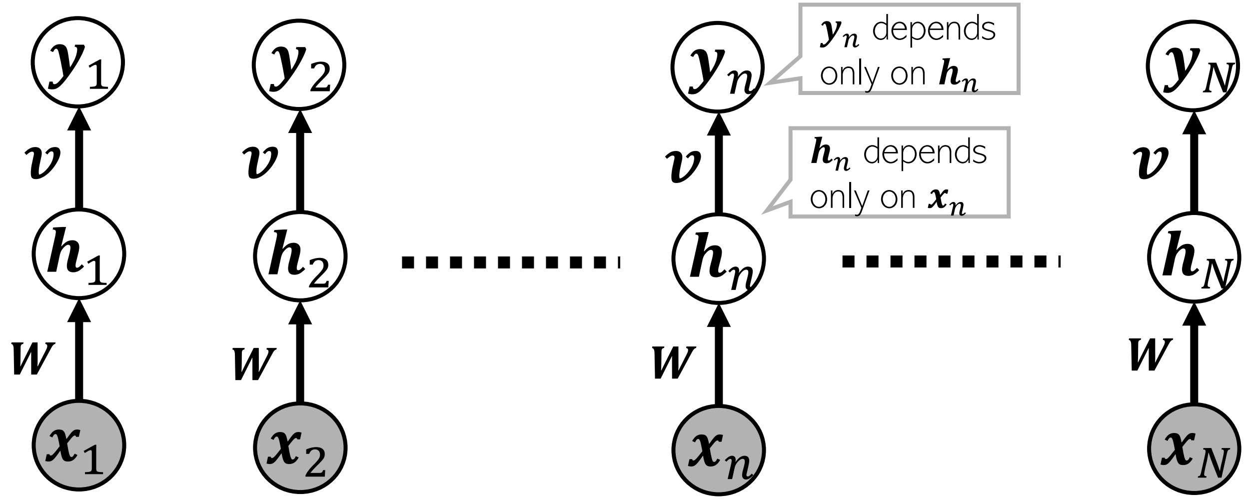
- In many problems, each input, each output, or both may be in form of sequences



- Different inputs or outputs need not have the same length
- Some examples of prediction tasks in such problems
  - Image captioning: Input is image (not a sequence), output is the caption (word sequence)
  - Document classification: Input is a word sequence, output is a categorical label
  - Machine translation: Input is a word sequence, output is a word sequence (in different language)
  - Stock price prediction: Input is a sequence of stock prices, output is its predicted price tomorrow
  - No input – just output (e.g., generation of random but plausible-looking text)

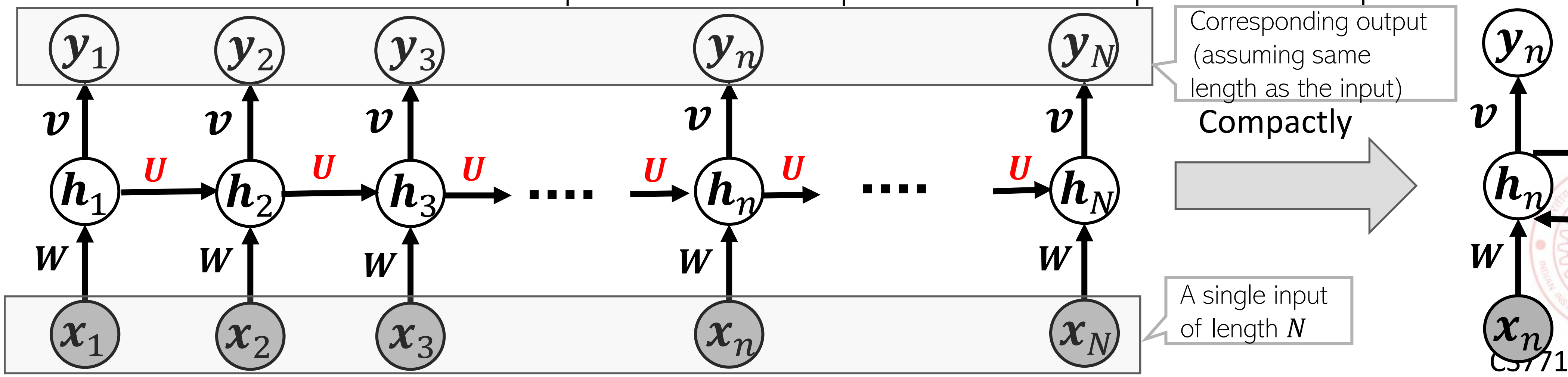
Recurrent Connections in Deep Neural Networks

- Feedforward nets such as MLP and CNN assume independent observations



Feedforward neural networks are not ideal when inputs $[\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N]$ and/or outputs $[\boldsymbol{y}_1, \boldsymbol{y}_2, \dots, \boldsymbol{y}_N]$ represent sequential data (e.g., sentences)

- A recurrent structure can be helpful if each input and/or output is a sequence



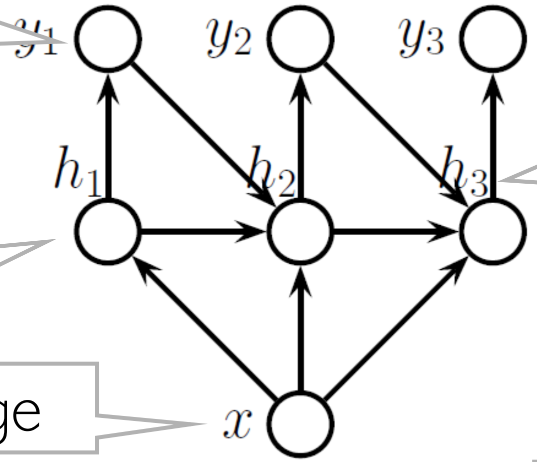
Recurrent Neural Networks: Some Examples

- Consider generating a sequence $y_1, y_2, \dots, y_T$ given an input x

Words in the generated caption

Hidden states at each step of the sequence

An image



Predicted y_{t-1} also fed into h_t

At test time, we can only feed the predicted y_{t-1}

During training, if the true y_{t-1} is fed, we call it "teacher forcing"

- Predicting the sentiment of a movie review

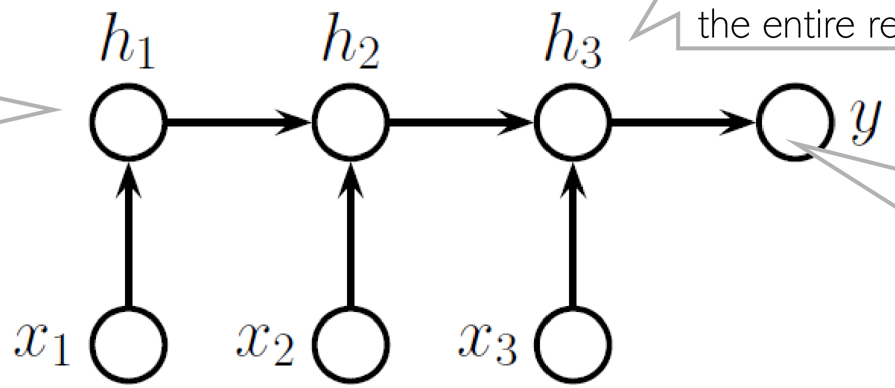
This final hidden state is supposed to contain the information about the entire review

Isn't this too much to expect?? ☺

Indeed; this can be an issue with RNNs

Hidden states

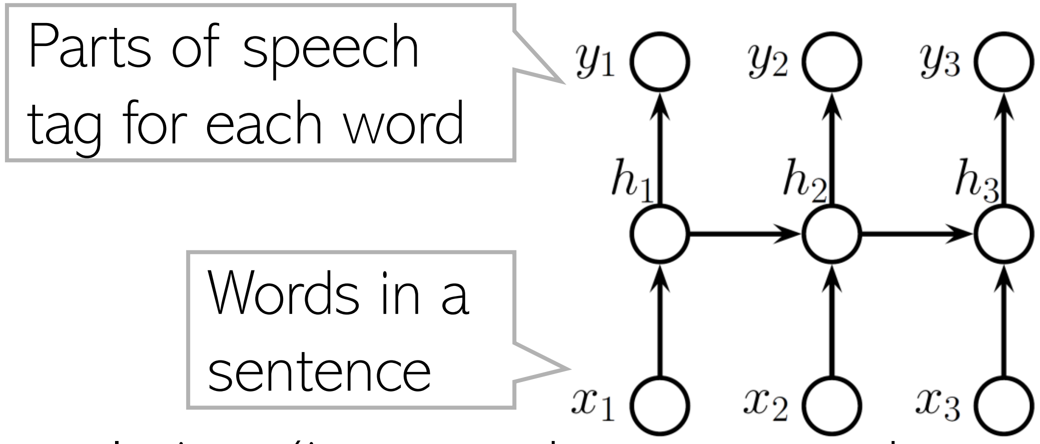
Words in the review



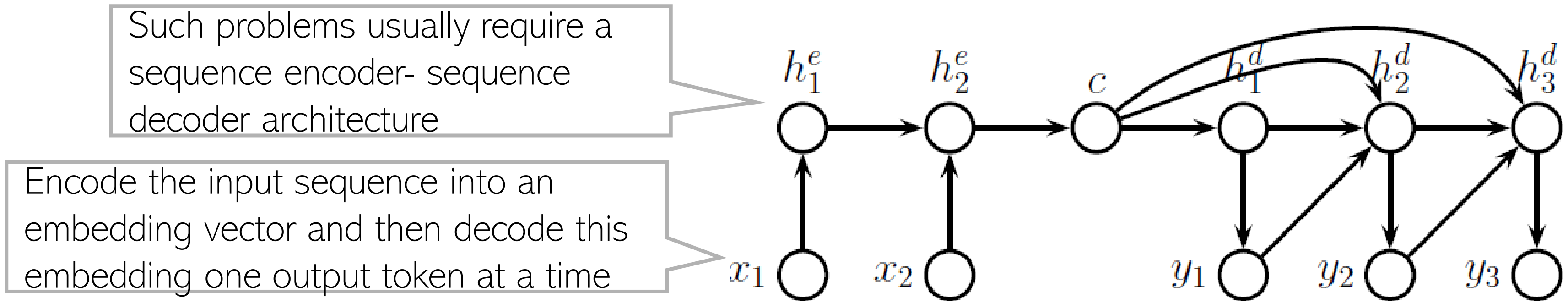
Predicted sentiment

Recurrent Neural Networks: Some Examples

- Parts of speech tagging (or "aligned" translation; input and output have same length)



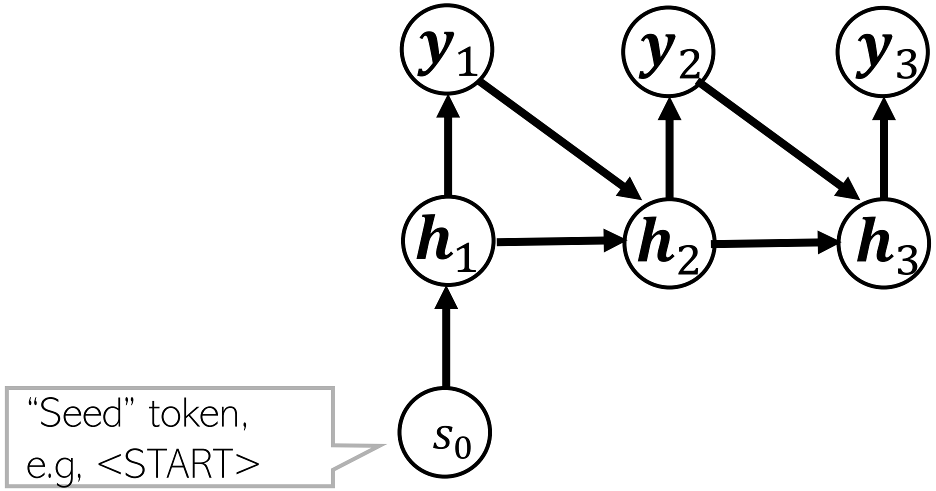
- "Unaligned" translation (input and output can have different lengths)



- In the unaligned case, generation stops when an "end" token (e.g., <END>) is generated on the output side

Recurrent Neural Networks: Some Examples

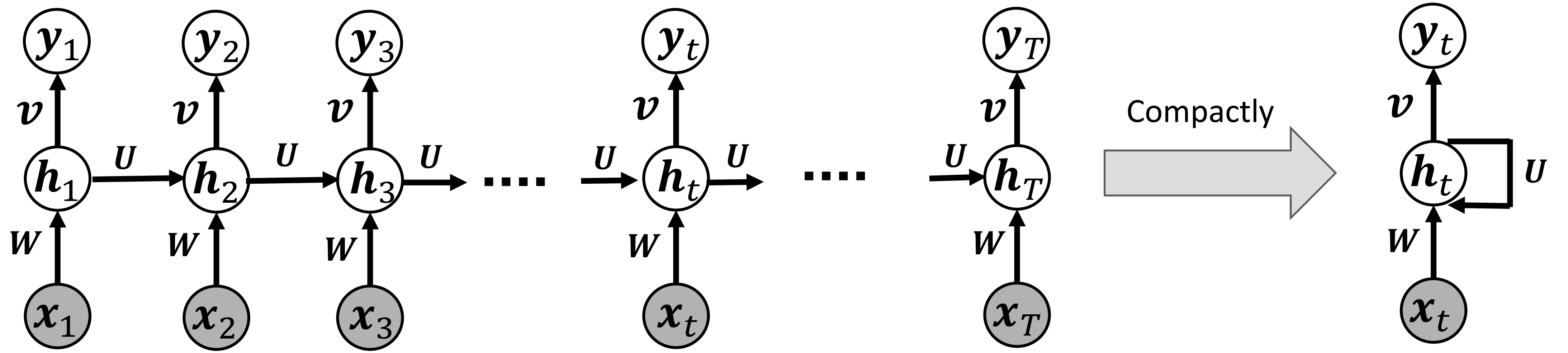
- Unconditional generation (no input, only an output sequence is generated given a RNN that was trained using some training data containing several sequences)



- Each generate word/token is fed to the next step's hidden state
- Generation stops when an "end" token (e.g., <END>) is generated

Recurrent Neural Networks

- A basic RNN's architecture (assuming input and output sequence have same lengths)



- RNN has three sets of weights $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{v}$
- $\boldsymbol{W}$ and $\boldsymbol{U}$ model how $\boldsymbol{h}_t$ at step t is computed: $\boldsymbol{h}_t = g(\boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{U}\boldsymbol{h}_{t-1})$

g is some activation function like ReLU

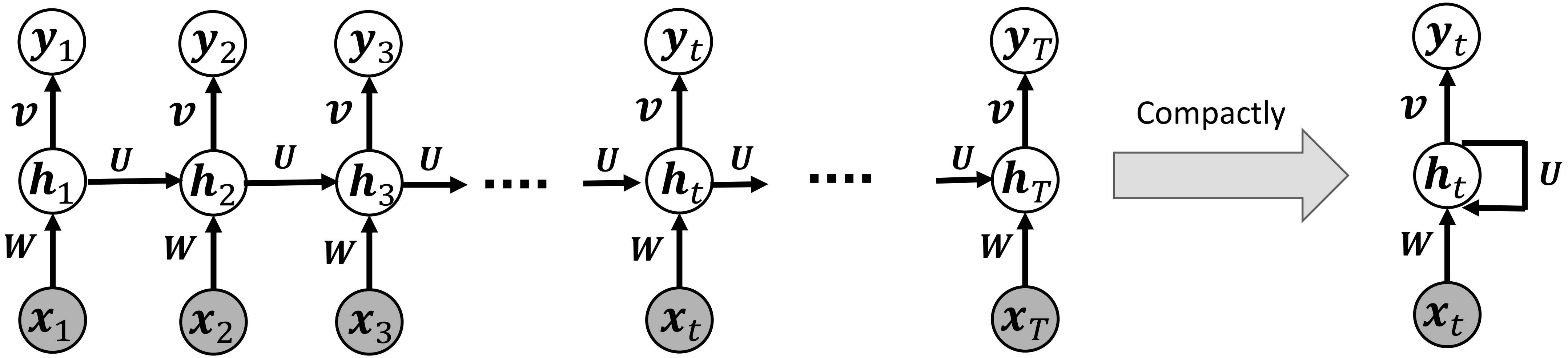
- $\boldsymbol{v}$ models the hidden layer to output mapping, e.g., $\boldsymbol{y}_t = o(\boldsymbol{v}\boldsymbol{h}_t)$

o depends on the nature of y_t.If it is categorical then o can be softmax

- Important: Same $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{v}$ are used at all steps of the sequence (weight sharing)

For RNNs, Long Distant Past is Hard to Remember

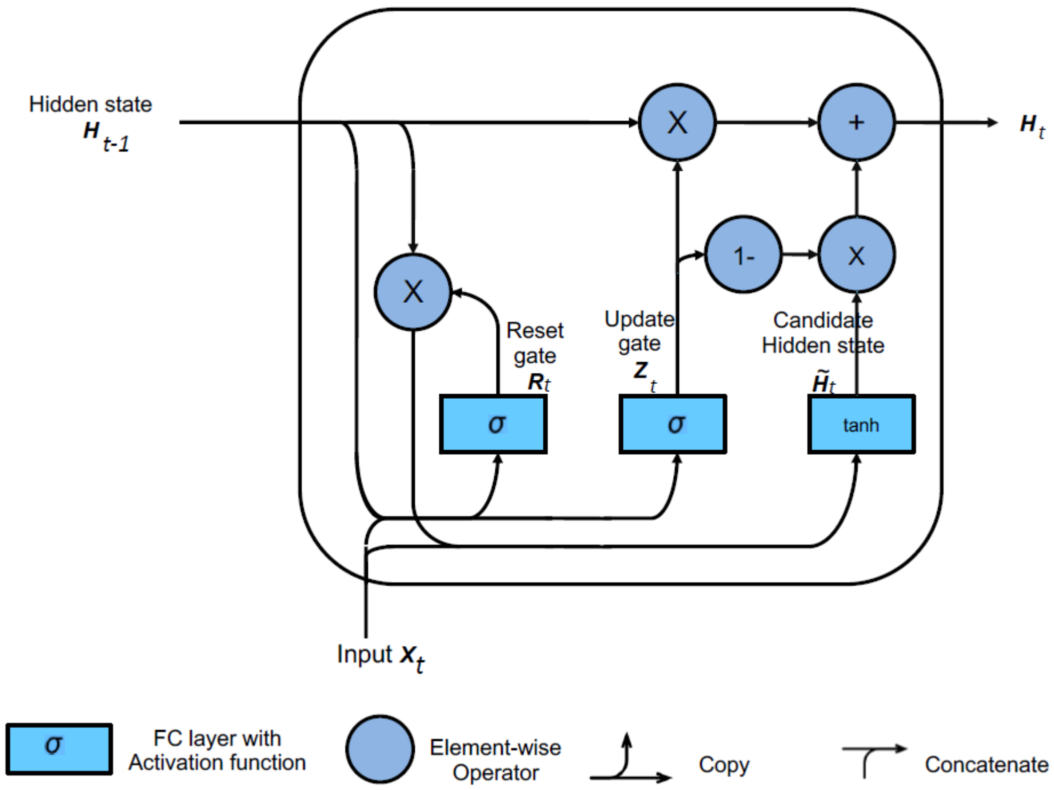
- The hidden layer nodes $\boldsymbol{h}_t$ are supposed to summarize the past up to time $\boldsymbol{t-1}$

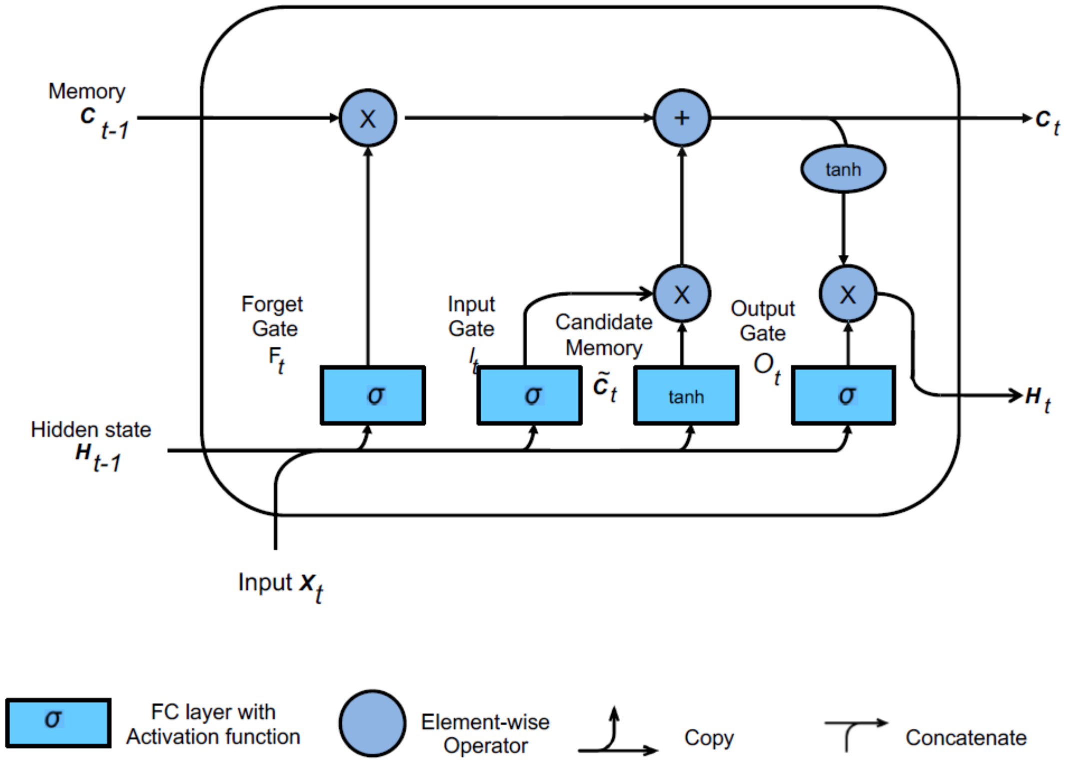


- In theory, they should. In practice, they can't. Some reasons
  - Vanishing gradients along the sequence too – past knowledge gets "diluted"
  - Hidden nodes also have limited capacity because of their finite dimensionality
- Various extensions of RNNs have been proposed to address forgetting
  - Gated Recurrent Units (GRU), Long Short Term Memory (LSTM)

GRU and LSTM

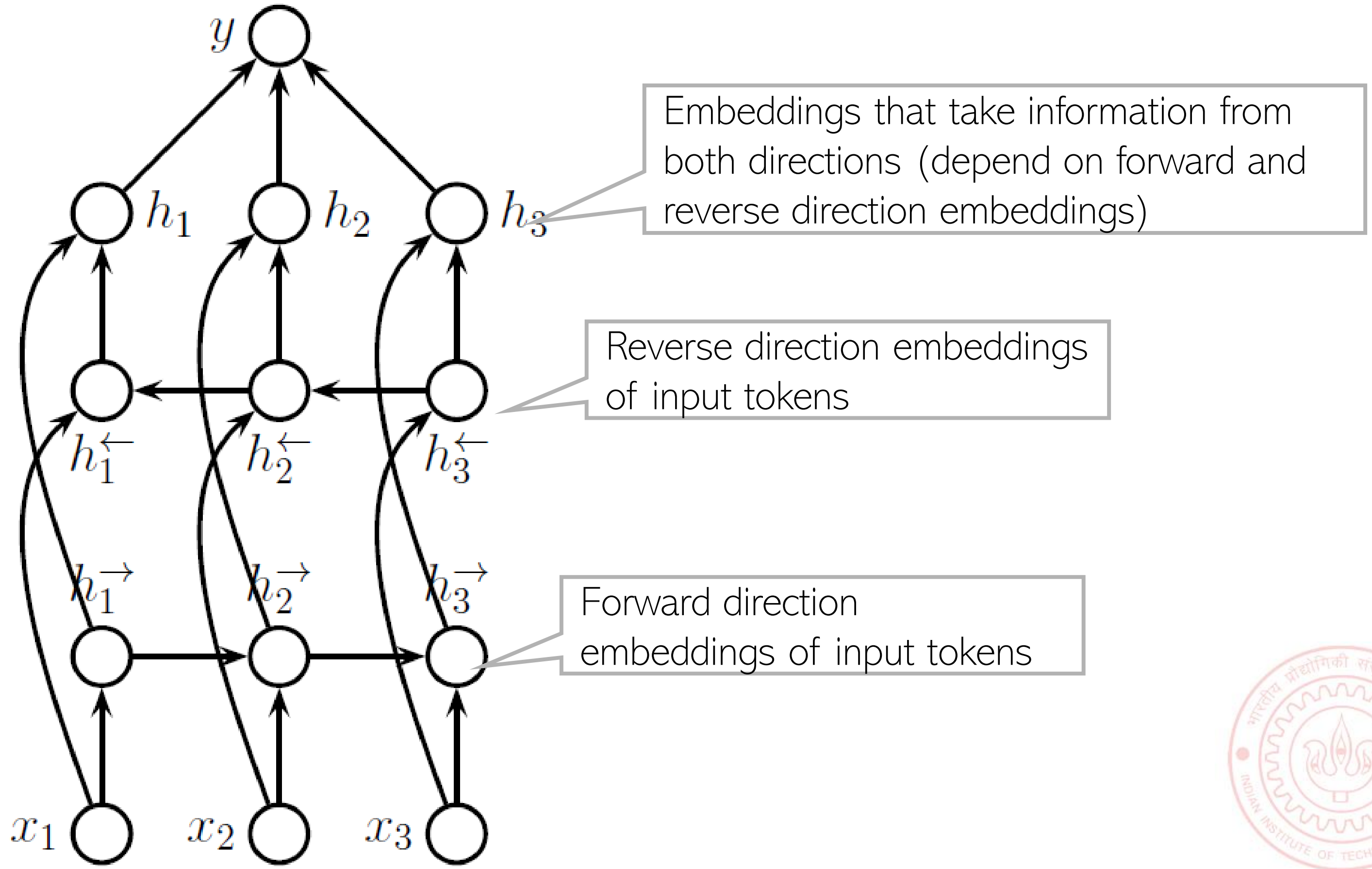
- GRU and LSTM are variants of RNNs. These contain specialized units and "memory" which modulate what/how much information from the past to retain/forget





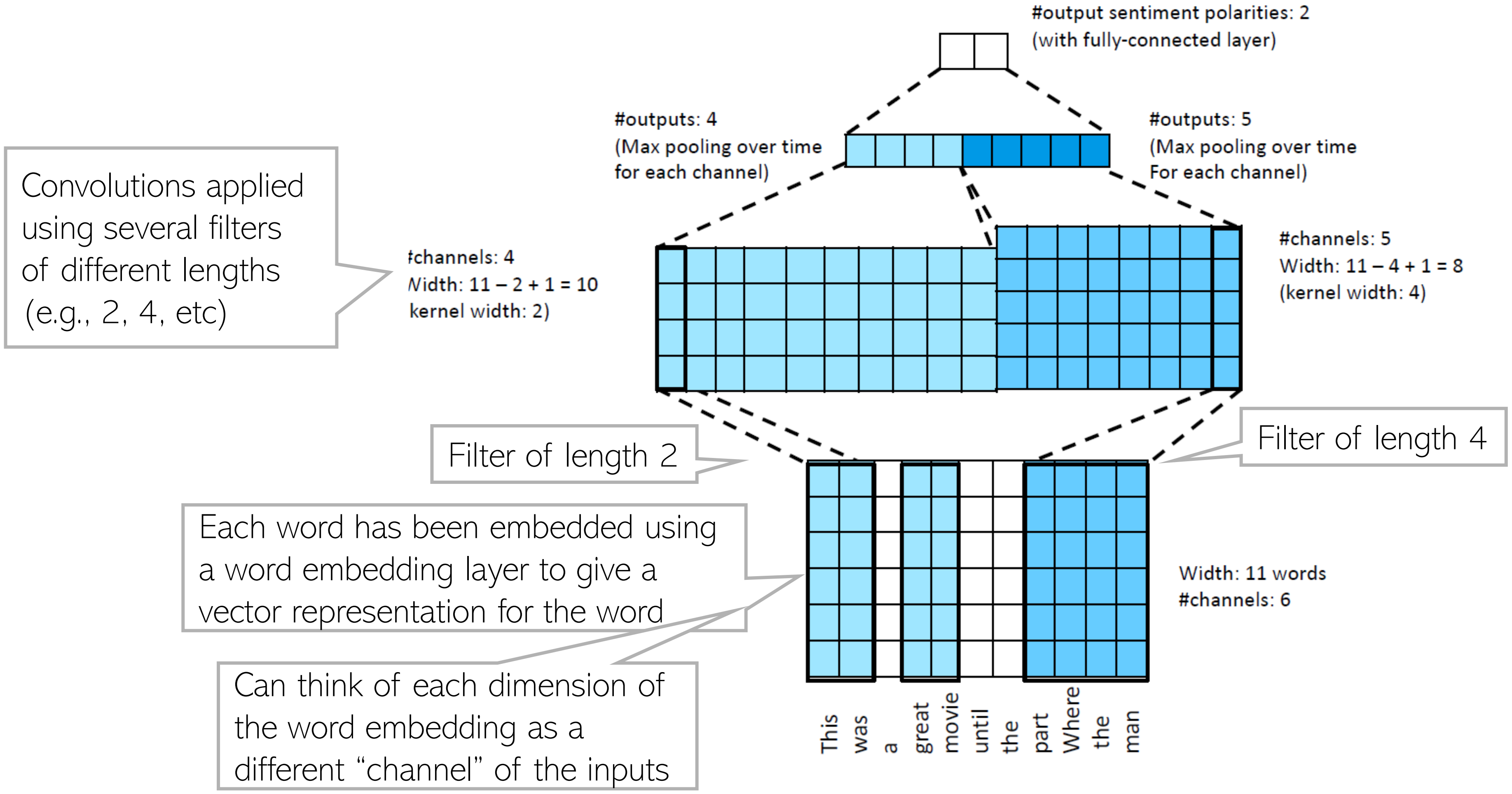
Bidirectional RNN

- RNNs and GRU and LSTM only remember the information from the previous tokens
- Bidirectional RNNs can remember information from the past and future tokens



CNN for Text

- CNNs can exploit sequential structure as well using convolutions
- Figure below is CNN for text data where the goal is to predict sentiment of a review



Need for Attention

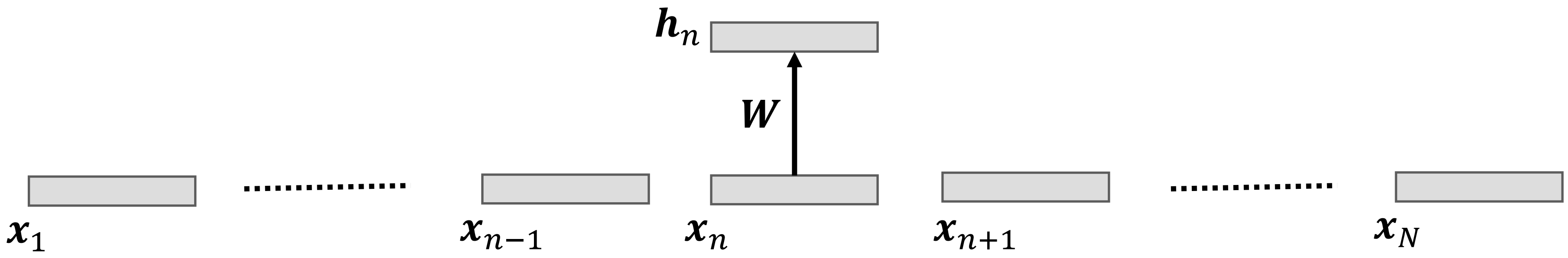
- Each layer in standard deep neural nets computes a linear transform + nonlinearity

- For N inputs, collectively denoting inputs as $\boldsymbol{X} \in \mathbb{R}^{N \times K_1}$ and outputs as $\boldsymbol{H} \in \mathbb{R}^{N \times K_2}$

$$\boldsymbol{H} = g(\boldsymbol{X}\boldsymbol{W})$$

Notation alert: Input $\boldsymbol{X}$ can be data (if $\boldsymbol{H}$ denotes first hidden layer) or the $\boldsymbol{H}$ of the previous hidden layer

- Here the weights $\boldsymbol{W} \in \mathbb{R}^{K_1 \times K_2}$ do not depend on the inputs $\boldsymbol{X}$
  - Output $\boldsymbol{h}_n = g(\boldsymbol{W}^\top \boldsymbol{x}_n) \in \mathbb{R}^{K_2}$ only depends on $\boldsymbol{x}_n \in \mathbb{R}^{K_1}$ and pays no attention to $\boldsymbol{x}_m$, $m \neq n$



  - When different inputs outputs have inter-dependencies (e.g., they denote representations of words in a sentence, or patches in an image), paying attention to other inputs is helpful/needed

Attention Mechanism

- Don't define output $\boldsymbol{h}_n$ as $\boldsymbol{h}_n = g(\boldsymbol{W}\boldsymbol{x}_n)$ but as a weighted combination of all inputs

$$\boldsymbol{h}_n = \sum_{i=1}^{N} \alpha_{ni}(\boldsymbol{X}) f(\boldsymbol{x}_i) = \sum_{i=1}^{N} \alpha_{ni}(\boldsymbol{X}) \boldsymbol{v}_i$$

α_{ni} is the attention score (to be learned) which tells us how much input $\boldsymbol{x}_i$ should attend to output $\boldsymbol{h}_n$

$\boldsymbol{v}_i$ is the "value" vector of input $\boldsymbol{x}_i$ (how input $\boldsymbol{x}_i$ should be used to compute the output $\boldsymbol{h}_n$)

- Attention scores $\alpha_{ni}(\boldsymbol{X})$ and "value" $\boldsymbol{v}_i = f(\boldsymbol{x}_i)$ of $\boldsymbol{x}_i$ can be defined in various ways

$N \times K$ matrix of "queries"

Row n is $\boldsymbol{q}_n$

Will be used to compute the attention scores for $\boldsymbol{h}_n$

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q$$

$N \times K$ matrix of N "keys"

Query $\boldsymbol{q}_n$ is compared with each of the N keys

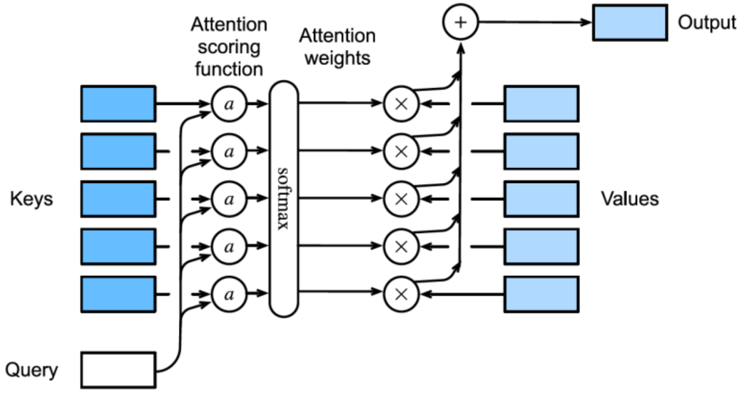
$$\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K$$

$N \times K$ matrix of "value" vectors of the N keys

$$\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V$$

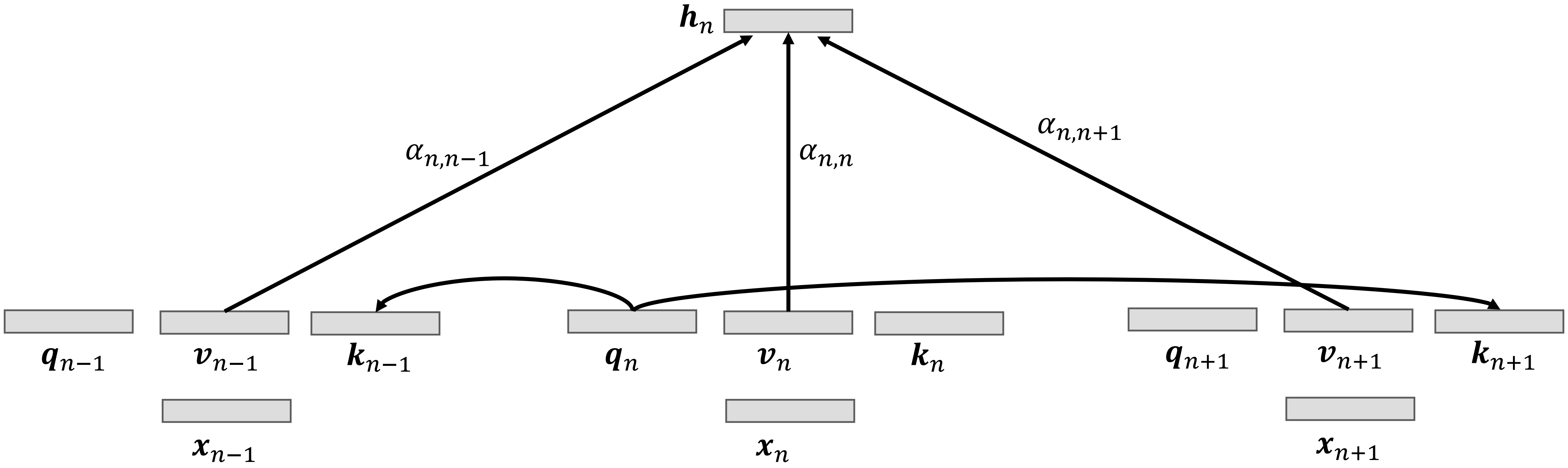
- One popular way to define the attention scores

$$\alpha_{ni}(\boldsymbol{X}) = \frac{\exp(\boldsymbol{q}_n^\top \boldsymbol{k}_i)}{\sum_{j=1}^{N} \exp(\boldsymbol{q}_n^\top \boldsymbol{k}_j)}$$



- Attention mechanism (especially self-attention is used in transformers)

Attention Mechanism



$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q$ $\quad$ $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K$ $\quad$ $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V$