Recurrent Architectures

Stephen Scott

(Adapted from Vinod Variyam and Ian Goodfellow)

sscott2@unl.edu

Introduction

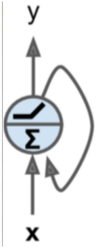
- All our architectures so far work on fixed-sized inputs
- Recurrent neural networks work on variable-length **sequences** of inputs
- E.g., text, biological sequences, video, audio
- Can also try 1D convolutions, but can lose long-term relationships in input
- Especially useful for NLP applications: translation, speech-to-text, sentiment analysis
- Can also **create novel output:** e.g., Shakespearean text, music

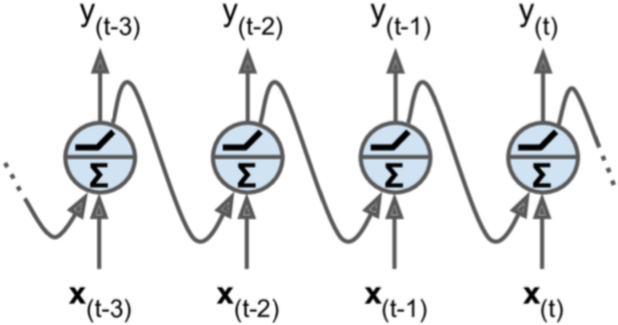
Outline

- Basic RNNs
- Input/Output Mappings
- Training
- Long short-term memory
- Gated Recurrent Unit
- Applications to natural language processing
  - Embedded representations
  - Bidirectional RNNs
  - Attention mechanisms
  - Transformers
- Summary

Basic Recurrent Cell

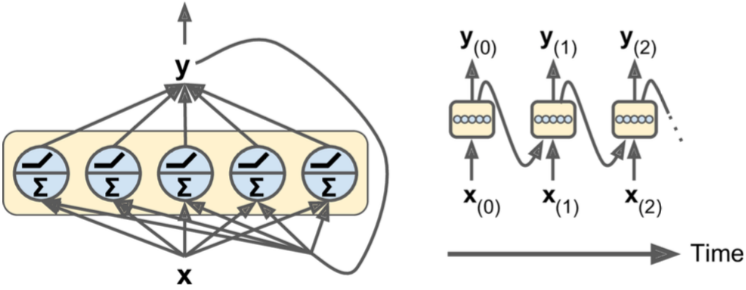
- A recurrent cell (or recurrent neuron) has connections pointing **backward** as well as forward
- At time step (frame) t, neuron receives input vector $\boldsymbol{x}_{(t)}$ as usual, but also receives its own output $y_{(t-1)}$ from previous step





Basic Recurrent Layer

- Can build a layer of recurrent cells, where each node gets both the vector $\boldsymbol{x}_{(t)}$ and the vector $\boldsymbol{y}_{(t-1)}$



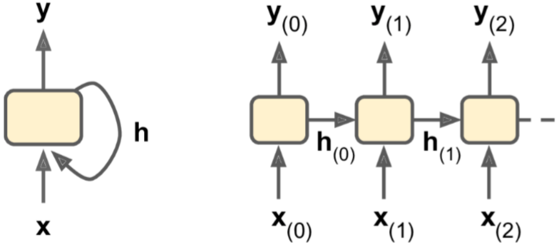
Basic Recurrent Layer

- Each node in the recurrent layer has independent weights for both $\boldsymbol{x}_{(t)}$ and $\boldsymbol{y}_{(t-1)}$
- For a single recurrent node, denote by $\boldsymbol{w}_{\boldsymbol{x}}$ and $\boldsymbol{w}_{\boldsymbol{y}}$
- For the entire layer, combine into matrices $W_{\boldsymbol{x}}$ and $W_{\boldsymbol{y}}$
- For activation function ϕ and bias vector $\boldsymbol{b}$, output vector is

$$\boldsymbol{y}_{(t)} = \phi\left(W_{\boldsymbol{x}}^{\top}\boldsymbol{x}_{(t)} + W_{\boldsymbol{y}}^{\top}\boldsymbol{y}_{(t-1)} + \boldsymbol{b}\right)$$

Memory and State

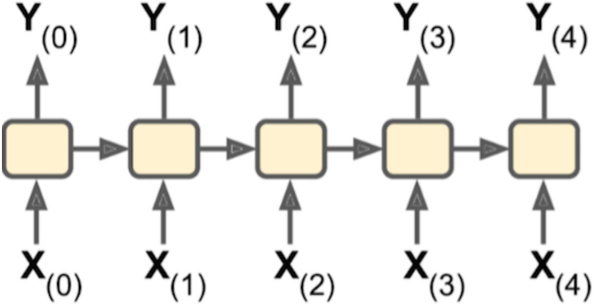
- Since a node's output depends on its past, it can be thought of having **memory** or **state**
- State at time t is $\boldsymbol{h}_{(t)} = f(\boldsymbol{h}_{(t-1)}, \boldsymbol{x}_{(t)})$ and output $\boldsymbol{y}_{(t)} = g(\boldsymbol{h}_{(t-1)}, \boldsymbol{x}_{(t)})$
- State could be the same as the output, or separate
- Can think of $\boldsymbol{h}_{(t)}$ as storing important information about input sequence
- Analogous to convolutional outputs summarizing important image features



Input/Output Mappings
Sequence to Sequence

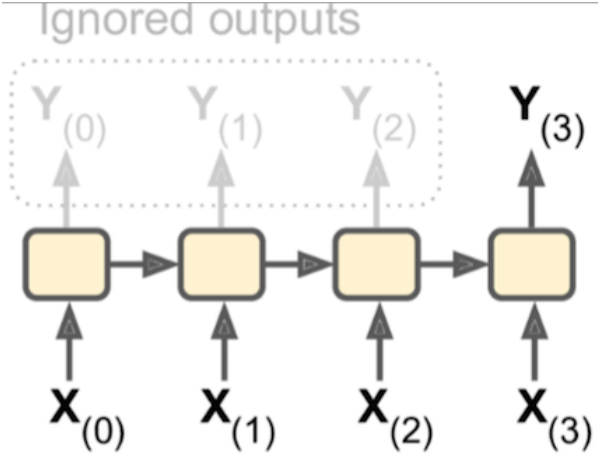
Many ways to employ this basic architecture:

- **Sequence to sequence:** Input is a sequence and output is a sequence
- E.g., series of stock predictions, one day in advance



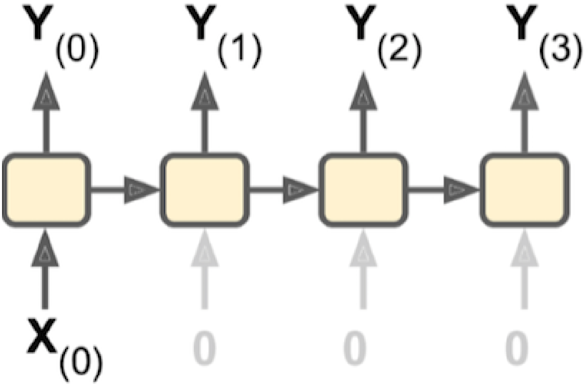
Input/Output Mappings
Sequence to Vector

- **Sequence to vector:** Input is sequence and output a vector/score/classification
- E.g., sentiment score of movie review

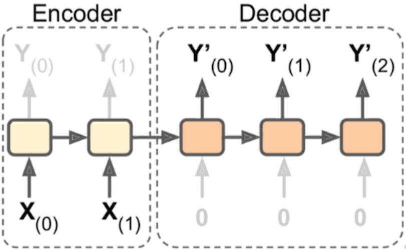


Input/Output Mappings
Vector to Sequence

- **Vector to sequence:** Input is a single vector (other times zeroes or repeated) and output is a sequence
- E.g., image to caption



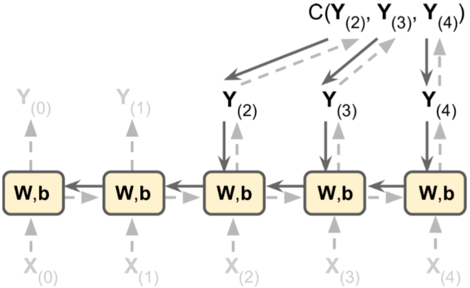
- **Encoder-decoder:** Sequence-to-vector (**encoder**) followed by vector-to-sequence (**decoder**)
- Input sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ yields hidden outputs $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T)$, then mapped to **context vector** $\boldsymbol{c} = f(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T)$
- Decoder output $\boldsymbol{y}_{t'}$ depends on previously output $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t'-1})$ and $\boldsymbol{c}$
- Example application: **neural machine translation**



Training

Backpropagation Through Time (BPTT)

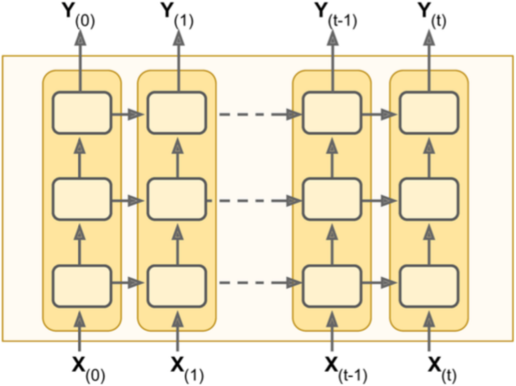
- Unroll through time and use BPTT
- Forward pass mini-batch of sequences through unrolled network yields output sequence $Y_{(0)}, \ldots, Y_{(T)}$
- Output sequence evaluated using **sequence loss** $C\left(\boldsymbol{Y}_{(0)}, \boldsymbol{Y}_{(1)}, \ldots, \boldsymbol{Y}_{(T)}\right)$, e.g., weighted sum of cross-entropies
- Gradients propagated backward through unrolled network (summing over all time steps), and parameters



- BPTT means that gradient is flowing through longer paths in graph $\Rightarrow$ **exploding** or **vanishing gradients**
  - Can happen with any network, but RNNs very susceptible
  - **Clipping** gradients to range $[-1, +1]$ can mitigate explosions

Deep RNNs

A **deep RNN** has multiple recurrent layers stacked

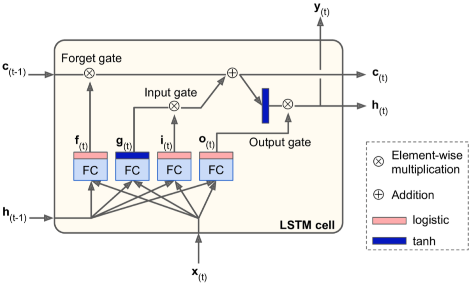


Training over Many Time Steps

- Vanishing and exploding gradients can be a problem with RNNs, like with other deep networks
  - Can maybe address with gradient clipping, etc.
- Can still suffer from long training times with long input sequences
  - **Truncated backpropagation through time** addresses this by limiting the number of unrolled training steps
  - Inhibits ability to learn long-term patterns
- In general, also have problem of first inputs of sequence have diminishing impact as sequence grows
  - E.g., first few words of long text sequence
- Goal: Introduce **long-term memory** to RNNs
- Allow a network to **accumulate** information about the past, but also decide when to **forget** information

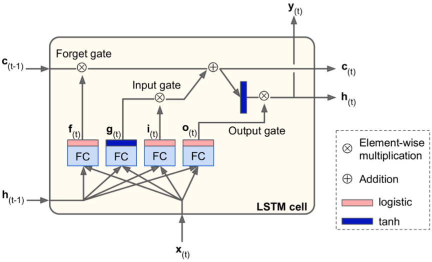
Long Short-Term Memory
Hochreiter and Schmidhuber (1997)

- Vector $\boldsymbol{h}_{(t)}$ = **short-term state**, $\boldsymbol{c}_{(t)}$ = **long-term state**
- At time t, some memories from $\boldsymbol{c}_{(t-1)}$ are forgotten in the **forget gate** and new ones (from **input gate**) added



- Result sent out as $\boldsymbol{c}_{(t)}$
- $\boldsymbol{h}_{(t)}$ (and $\boldsymbol{y}_{(t)}$) comes from processing long-term state in **output gate**

Long Short-Term Memory
Hochreiter and Schmidhuber (1997)

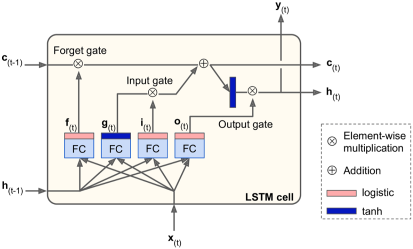


- $\boldsymbol{g}_{(t)}$ combines input $\boldsymbol{x}_{(t)}$ with state $\boldsymbol{h}_{(t-1)}$
- $\boldsymbol{f}_{(t)}, \boldsymbol{i}_{(t)}, \boldsymbol{o}_{(t)}$ are **gate controllers**
- $\boldsymbol{f}_{(t)} \in [0,1]^n$ controls forgetting of $\boldsymbol{c}_{(t-1)}$
- $\boldsymbol{i}_{(t)}$ controls remembering of $\boldsymbol{g}_{(t)}$
- $\boldsymbol{o}_{(t)}$ controls what of $\boldsymbol{c}_{(t)}$ goes to output and $\boldsymbol{h}_{(t)}$
- Output depends on long- and short-term memory
- Network learns what to remember long-term based on $\boldsymbol{x}_{(t)}$ and $\boldsymbol{h}_{(t-1)}$

Long Short-Term Memory
Hochreiter and Schmidhuber (1997)

- $\boldsymbol{i}_{(t)} = \sigma\left(W_{\boldsymbol{xi}}^{\top}\boldsymbol{x}_{(t)} + W_{\boldsymbol{hi}}^{\top}\boldsymbol{h}_{(t-1)} + \boldsymbol{b_i}\right)$
- $\boldsymbol{f}_{(t)} = \sigma\left(W_{\boldsymbol{xf}}^{\top}\boldsymbol{x}_{(t)} + W_{\boldsymbol{hf}}^{\top}\boldsymbol{h}_{(t-1)} + \boldsymbol{b_f}\right)$
- $\boldsymbol{o}_{(t)} = \sigma\left(W_{\boldsymbol{xo}}^{\top}\boldsymbol{x}_{(t)} + W_{\boldsymbol{ho}}^{\top}\boldsymbol{h}_{(t-1)} + \boldsymbol{b_o}\right)$
- $\boldsymbol{g}_{(t)} = \tanh\left(W_{\boldsymbol{xg}}^{\top}\boldsymbol{x}_{(t)} + W_{\boldsymbol{hg}}^{\top}\boldsymbol{h}_{(t-1)} + \boldsymbol{b_g}\right)$

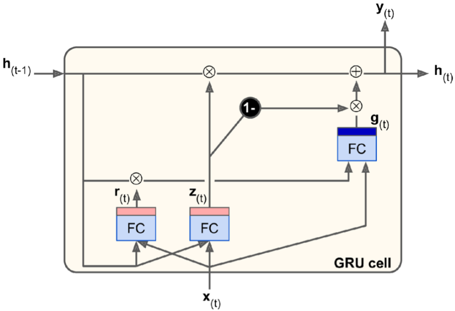
- $\boldsymbol{c}_{(t)} = \boldsymbol{f}_{(t)} \otimes \boldsymbol{c}_{(t-1)} + \boldsymbol{i}_{(t)} \otimes \boldsymbol{g}_{(t)}$
- $\boldsymbol{y}_{(t)} = \boldsymbol{h}_{(t)} = \boldsymbol{o}_{(t)} \otimes \tanh\left(\boldsymbol{c}_{(t)}\right)$



- Can add **peephole connection:** Let $\boldsymbol{c}_{(t-1)}$ affect $\boldsymbol{f}_{(t)}$ and $\boldsymbol{i}_{(t)}$ and $\boldsymbol{c}_{(t)}$ affect $\boldsymbol{o}_{(t)}$

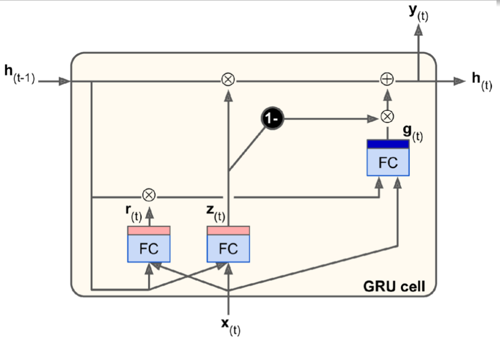
Gated Recurrent Unit

- Simplified LSTM
- Merge $\boldsymbol{c}_{(t)}$ into $\boldsymbol{h}_{(t)}$
- Merge $\boldsymbol{f}_{(t)}$ and $\boldsymbol{i}_{(t)}$ into $z_{(t)}$
  - $z_{(t),i} = 0 \Rightarrow$ forget $\boldsymbol{h}_{(t-1),i}$ and add in $\boldsymbol{g}_{(t),i}$



- $\boldsymbol{o}_{(t)}$ replaced by $\boldsymbol{r}_{(t)} \Rightarrow$ forget part of $\boldsymbol{h}_{(t-1)}$ when computing $\boldsymbol{g}_{(t)}$

Gated Recurrent Unit



- $z_{(t)} = \sigma\left(W_{xz}^\top \boldsymbol{x}_{(t)} + W_{hz}^\top \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_z\right)$
- $\boldsymbol{r}_{(t)} = \sigma\left(W_{xr}^\top \boldsymbol{x}_{(t)} + W_{hr}^\top \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_r\right)$
- $\boldsymbol{g}_{(t)} = \tanh\left(W_{xg}^\top \boldsymbol{x}_{(t)} + W_{hg}^\top \left(\boldsymbol{r}_{(t)} \otimes \boldsymbol{h}_{(t-1)}\right) + \boldsymbol{b}_g\right)$
- $\boldsymbol{y}_{(t)} = \boldsymbol{h}_{(t)} = z_{(t)} \otimes \boldsymbol{h}_{(t-1)} + \left(\boldsymbol{1} - z_{(t)}\right) \otimes \boldsymbol{g}_{(t)}$

- Common application of sequential models such as RNNs
- Example problems:
  - Sentiment analysis
  - Information extraction
  - Question answering
  - Summarization
  - **Machine translation**
- Some tasks involve classification, some text generation

Application: Natural Language Processing
Outline

- Embedded representations of words
- Neural machine translation (NMT) via encoder-decoder architecture
- Bidirectional RNNs
- Attention mechanisms
- Transformer architectures

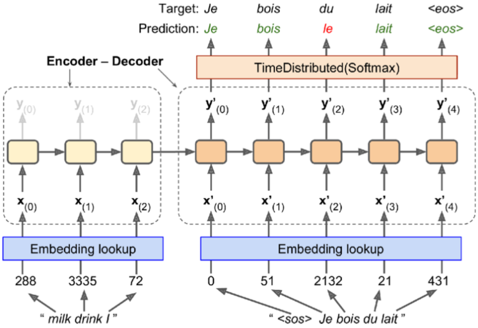
Embedded Representations of Words

- To process text with RNN, need **numeric** representation of words
- Simple approach: **Sparse vectors**, using a **one-hot** vector per word
  - $|V|$-dimensional vector, where $V =$ vocabulary
  - Represent document with disjunction of its words' vectors $\Rightarrow$ **bag-of-words**
- **Issues with this:**
  - Very high-dimensional inputs ($|V| > 10^5$)
  - All word vectors orthogonal $\Rightarrow$ don't cluster
- Alternative: **Dense vector** representation where similar words near each other
- Train embedding as part of RNN or use pre-trained one
  - E.g., **word2vec**

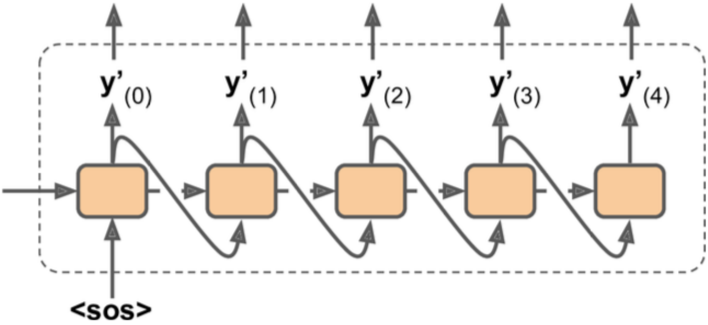
NMT via Encoder-Decoder Architecture

Training

- Pre-trained word embeddings fed into input
  - Both input and translation (shifted back one time step)
  - Each time step is classification ⇒ use cross-entropy
- Encoder maps word sequence to vector, decoder maps to translation via softmax distribution

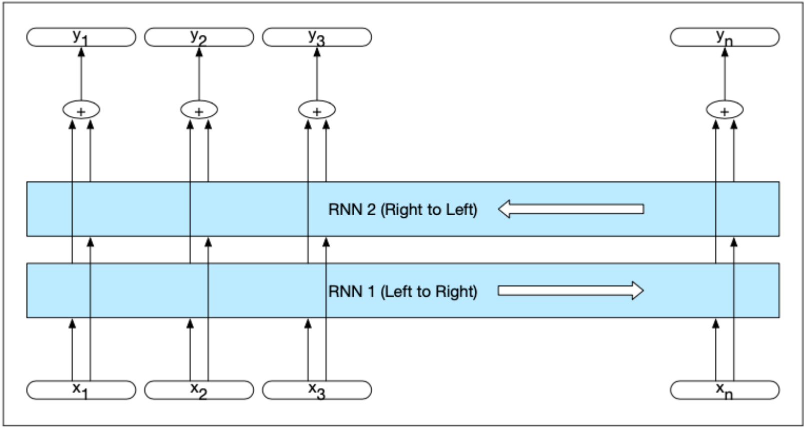


- After training, do translation by feeding previous translated word $y'_{(t-1)}$ to decoder



Bidirectional RNNs

- In addition to working from start to end, can simultaneously process from end towards start
- Two separate RNNs run in opposite directions and outputs are concatenated, added, etc.

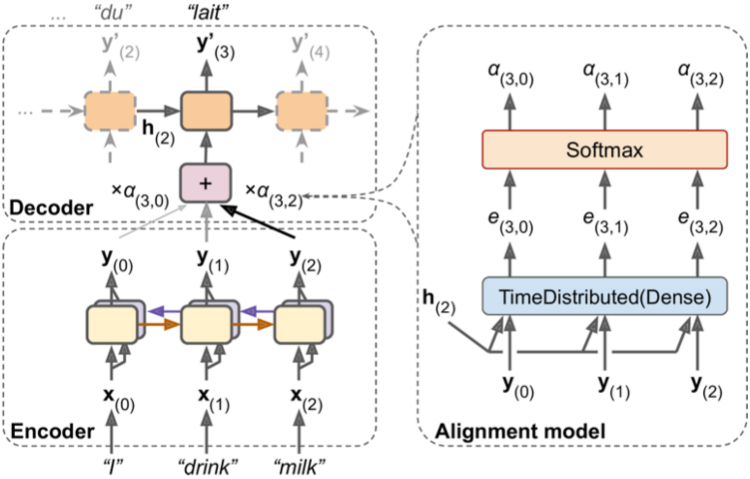


Attention Mechanisms

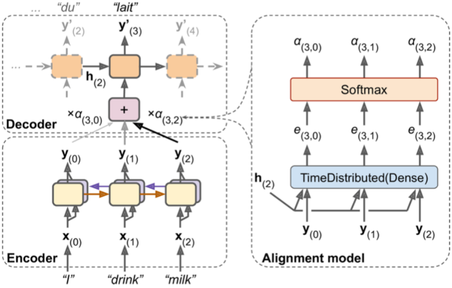
- Encoder-decoder works through an **embedded space** like an autoencoder, so can represent the entire input as an embedded vector prior to decoding
- **Issue:** Need to ensure that the context vector fed into decoder is sufficiently large in dimension to represent context required
- Can address this representation problem via **attention mechanism**
  - Encodes input sequence into a **vector sequence** rather than single vector
  - As it decodes translation, decoder focuses on relevant subset of the vectors

Attention Mechanisms

- At time step t, decoder weights encoder output i by $\alpha_{(t,i)}$
- E.g., $t = 3$ has $\alpha_{(3,2)} > \alpha_{(3,0)}, \alpha_{(3,1)}$ since "lait" depends more on "milk" than other words



- Weights $\alpha_{(t,i)}$ come from **alignment model** trained at same time as encoder-decoder
- Input is encoder outputs $(\boldsymbol{y}_{(0)}, \boldsymbol{y}_{(1)}, \ldots)$ and decoder hidden state $(\boldsymbol{h}_{(2)})$
- Output is **energy** score $e_{(t,i)}$ measuring how well output $\boldsymbol{y}_{(i)}$ aligns with $\boldsymbol{h}_{(t-1)}$ (or $\boldsymbol{h}_{(t)}$)
  - I.e., how much one should focus on word encoding $\boldsymbol{y}_{(i)}$ when generating output $\boldsymbol{y}'_{(t)}$



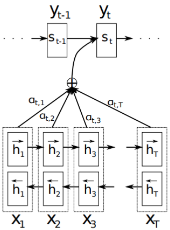
Attention Mechanisms
Calculations

$$\tilde{\boldsymbol{h}}_{(t)} = \sum_i \alpha_{(t,i)} \boldsymbol{y}_{(i)}$$

$$\alpha_{(t,i)} = \frac{\exp\left(e_{(t,i)}\right)}{\sum_{i'} \exp\left(e_{(t,i')}\right)}$$

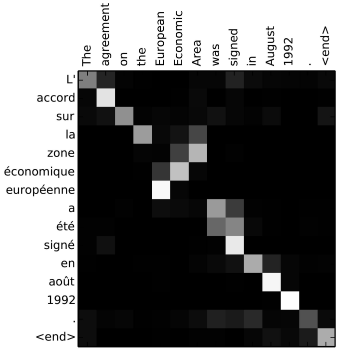
$$e_{(t,i)} = \begin{cases} \boldsymbol{h}_{(t)}^\top \boldsymbol{y}_{(i)} & \text{dot product} \\ \boldsymbol{h}_{(t)}^\top W \boldsymbol{y}_{(i)} & \text{bilinear} \\ \boldsymbol{v}^\top \tanh\left(W\left[\boldsymbol{h}_{(t)}; \boldsymbol{y}_{(i)}\right]\right) & \text{concatenation and activation} \end{cases}$$

$\boldsymbol{y}$ vectors can also be concatenation from bidirectional RNNs



Attention Mechanisms

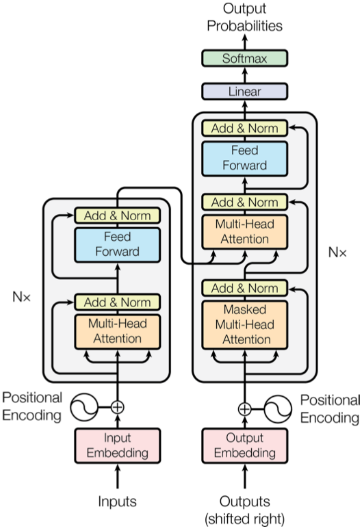
- The ith element of **attention vector** α_t tells us the probability that target output y'_i is aligned to (or translated from) input x_t
- Then $\tilde{h}_i$ is expected annotation over all annotations with probabilities α_t



Transformer Architecture (Vaswani et al., 2017)

- State-of-the-art performance in NMT **without** recurrent layers; only attention
  - Feed inputs sequentially as before, but no memory
  - Faster to train
- Basis of cutting-edge approaches such as **BERT** and **GPT-3**

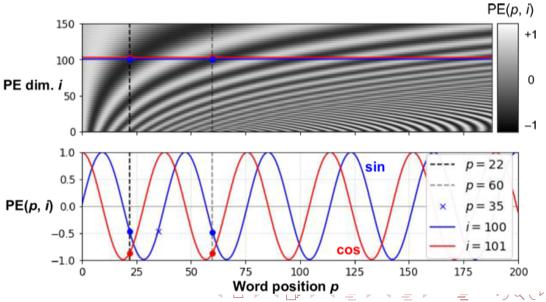
- Encoder on left is stack of N **multi-head attention** and feed-forward layers (no recurrent)
- Decoder on right takes outputs and encoder outputs and predicts distribution over next word
- Replace memory with **positional encoding**
- Training and inference similar to RNN



- **Positional encoding** vector added to word embedding vector (different sum for same word in different spots)
- Matrix P such that $P_{p,i}$ is ith component of embedding for pth word in sentence ($d = 512$)

$$P_{p,2i} = \sin\left(p/10000^{2i/d}\right) \quad P_{p,2i+1} = \cos\left(p/10000^{2i/d}\right)$$

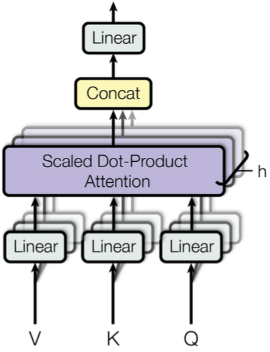
22nd word positional encoding is vector with ≈ -0.488 in 100th position and ≈ -0.873 in 101st position



Transformer Architecture (Vaswani et al., 2017)

Multi-Head Attention

- Generally, attention can be thought of as matching a **query** to a **key**, mapping to a distribution, then scaling a set of **values**
  - E.g., from earlier, $\boldsymbol{h}_{(t)}$ is query, $\boldsymbol{y}_{(i)}$ is key



- Work with batches to use **scaled dot-product attention**:

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

d_k = number of keys

- **Multi-head attention** defines multiple such matrices and concatenates outputs

Transformer Application: GPT-3

- Initially built via unsupervised pretraining of transformer modules, then fine-tuned on various NLP tasks
- 175 billion parameters (!!!)
- Such realistic generation of fake news articles, etc., creators warn of potential "harmful effects"

Transformer Application: BERT

Bidirectional Encoder Representations from Transformers

- An embedding that depends on the **context** of the word in the sentence
  - "We went to the river **bank**"
  - "I need to go to the **bank** to make a deposit"
- In contrast to **context-free models** (word2vec, GloVe)
- BERT trained on entire sentences by **masking** out random words and predicting them

  $\Rightarrow$ **Self-supervised training**

Summary

- **Recurrent neural networks** process **sequential data**
- At time t an RNN generates output based on input at time t and its **state** (or memory) at time $t-1$
- RNNs are trained by **unrolling through time** the computation graph and updating with backpropagation
- **LSTMs** and **GRUs** maintain **long-term** memory, which allows for more robust models and easier training
- **Bidirectional** RNNs process from both the beginning of a sequence to the end, and from end to beginning
- Performance is further enhanced via **attention mechanisms** that allow a decoder to focus on different parts of the input at different times
- **Transformer architectures** are non-recurrent, attention-based neural approaches that are the state of the art in NLP