CS 224S / LINGUIST 285
Spoken Language Processing

Andrew Maas
Stanford University
Spring 2022

Lecture 5: Deep Learning Preliminaries

Original slides by Mike Wu

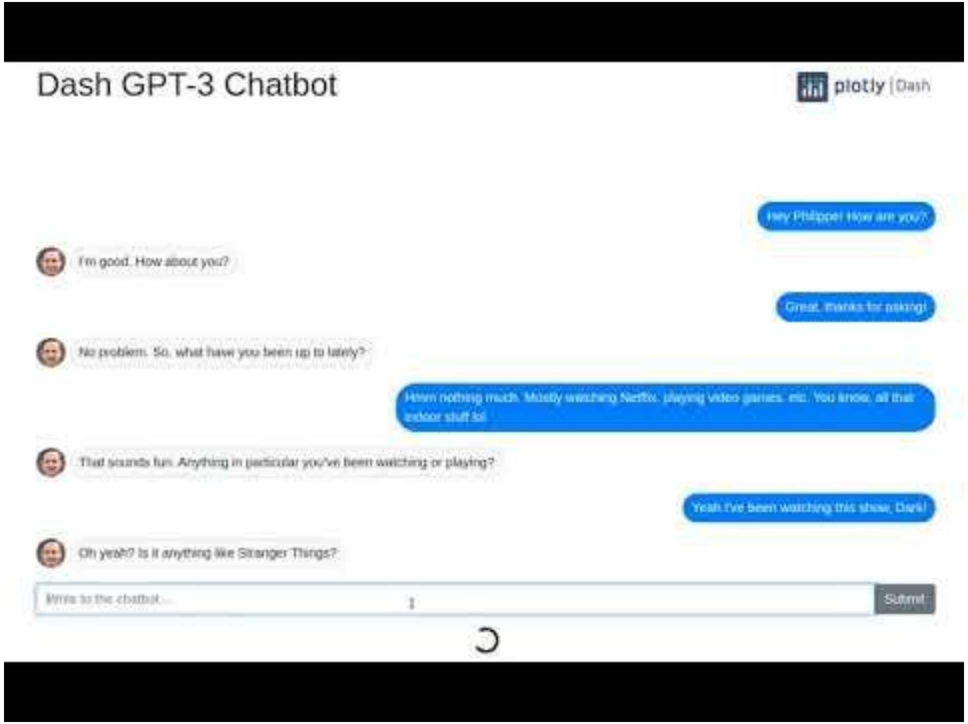
Outline for today

HW1 due now. HW2 posted to course website tonight

- Neural network basics: Hidden layers and gradient computation
- Recurrent NNs
  - Introduction
  - Encoder-decoder
- Attention
  - Introduction
  - Multi-headed attention
- Transformers: stacking attention

Chatbots in the wild

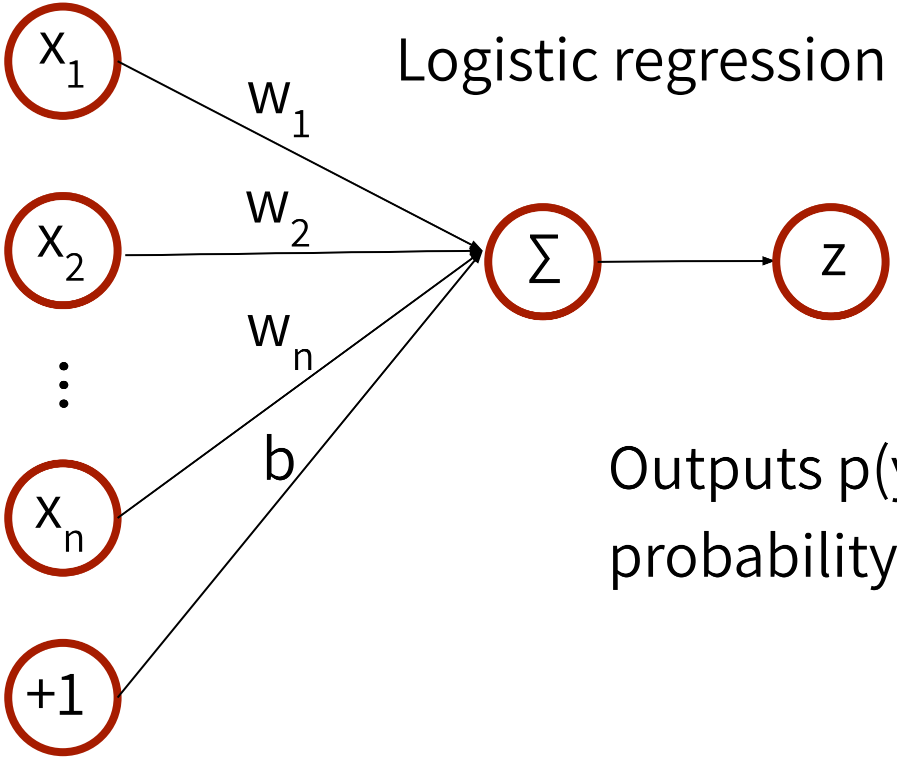
Large encoder decoder transformer models are doing wonders for realistic chatbots.



Neural Network Review

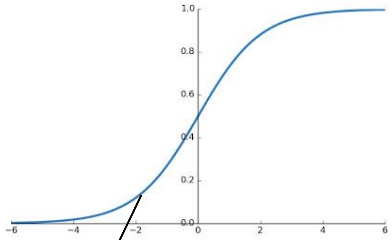
Logistic Regression

Logistic regression as a “neuron’



$= \sum w_i x_i + b$

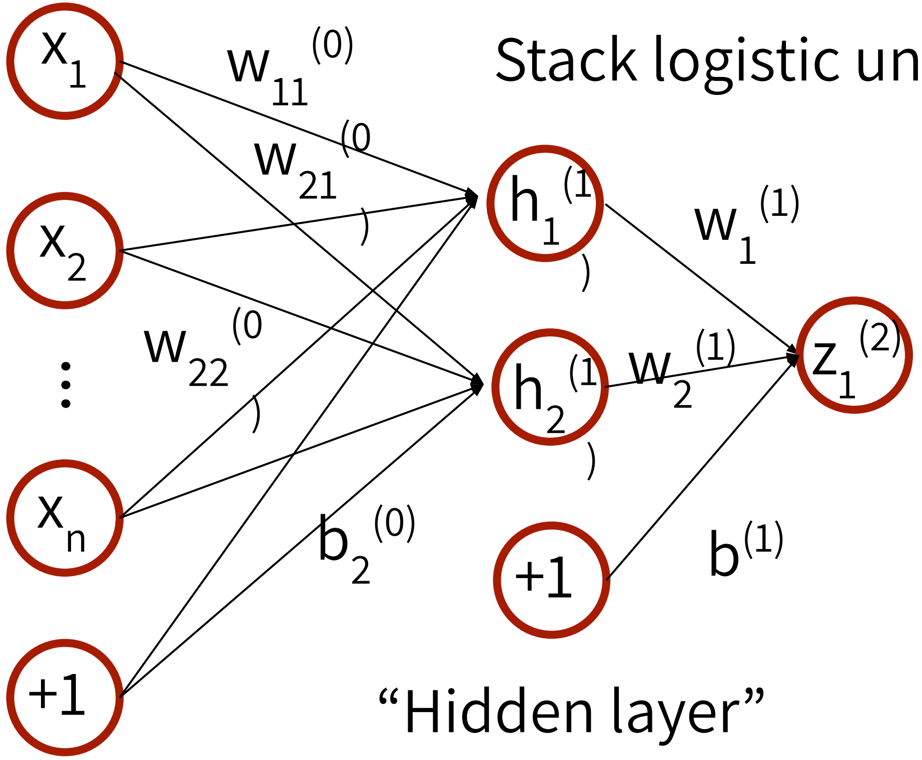
nonlinearity



Outputs $p(y=1|x_1 \ldots x_n) = \sigma(z)$, the probability of predicting class 1!

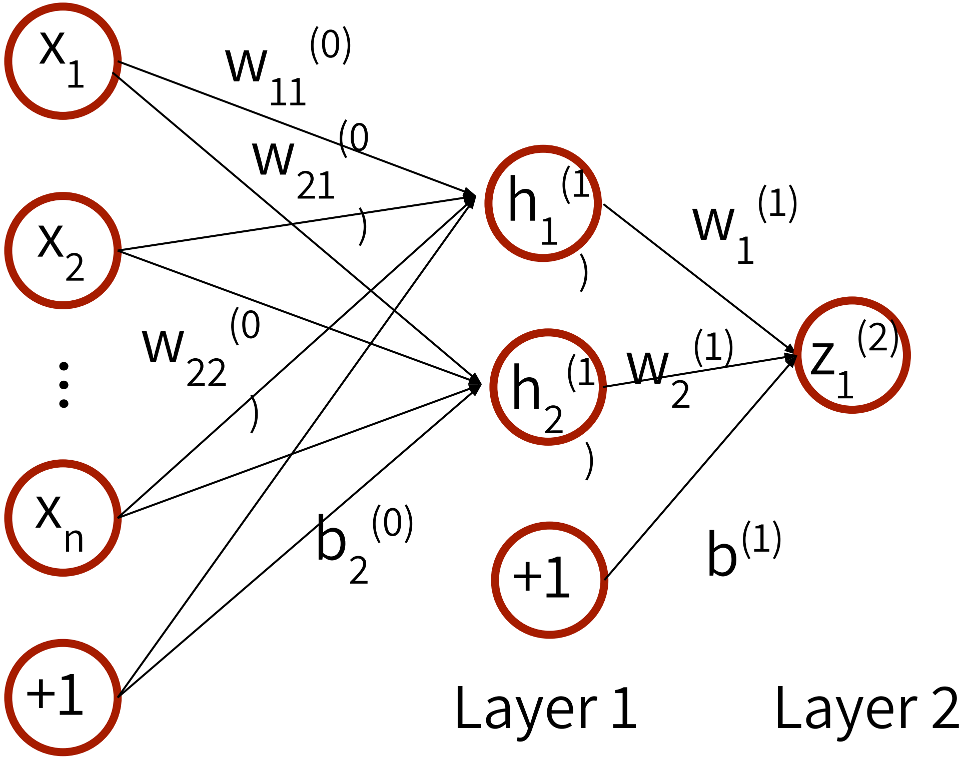
Multi-layer Perceptrons

Stack logistic units!



“Hidden layer”

Multi-layer Perceptrons



<u>Forward Propagation</u>

$z_j^{(l+1)} = \sum w_{ij}^{(l)} a_i^{(l)} + b_j^{(l)}$

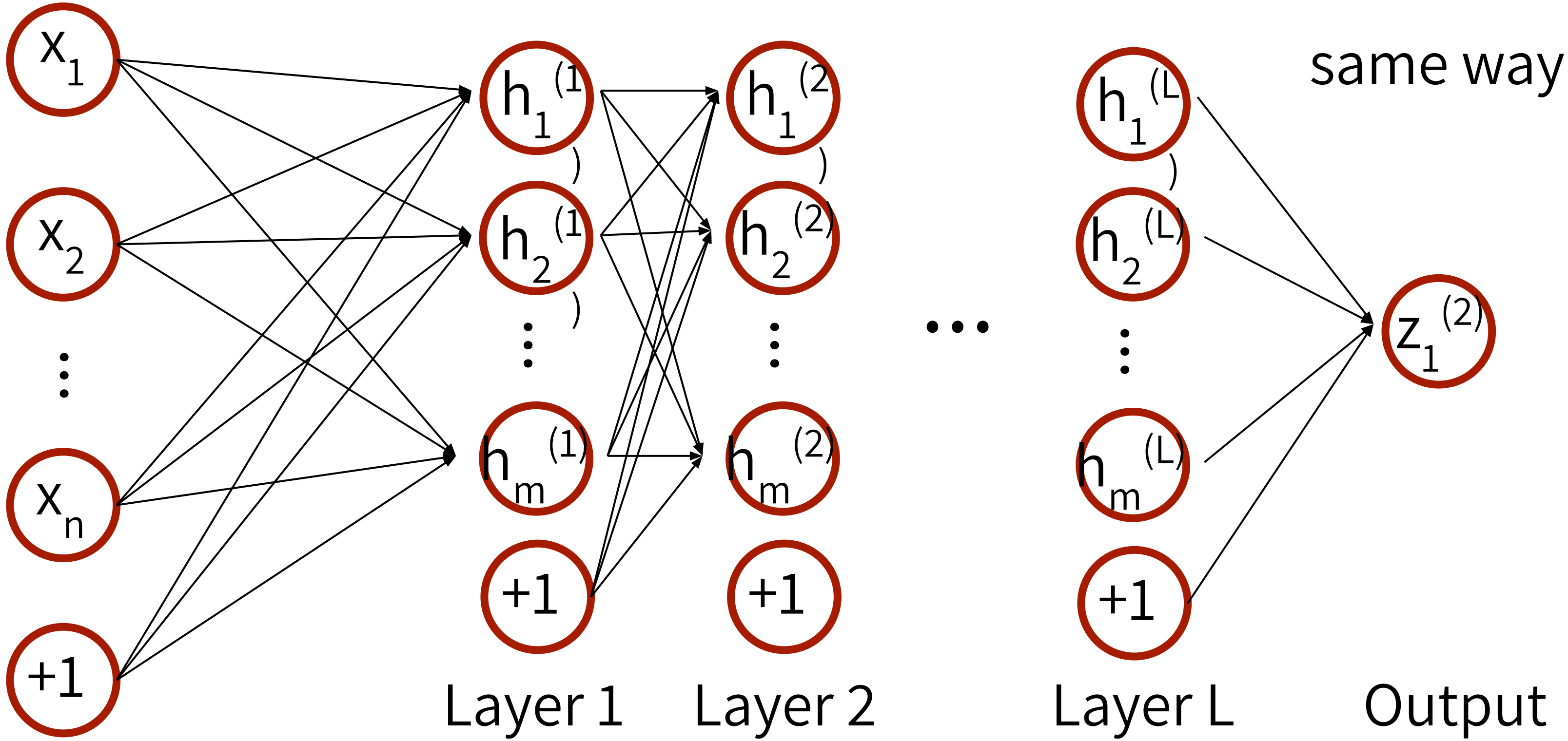
$h_j^{(l)} = \sigma(z_j^{(l)})$ "*activation*"

$\theta = \{ w_{ij}^{(l)} , b_j^{(l)}$ for all i,j,l$\}$

^ "parameters"

“Deep” Neural Networks

Forward pass works the same way



Objective Function

Depends on the task! Examples:

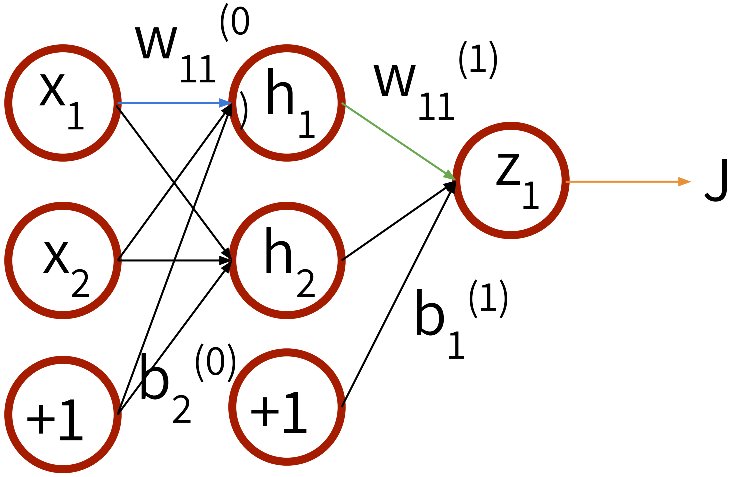
Binary classification	Multiclass classification	Regression
Label: $y \in \{0,1\}$	Label: $y \in \{1,...,K\}$	Label: $y \in R^d$
Objective:	Objective:	Objective:
$p = \text{sigmoid}(z)$	$p_{1:K} = [p_1, \ldots, p_k] = \text{softmax}(z_{1:K})$	$J_\theta = \text{sum}(\text{sqrt}(z - y)) / d$
$J_\theta = y \log p + (1-y) \log(1-p)$	$J_\theta = -\sum y_c^{\text{onehot}} \log p_c$	

Now we have to optimize!

Backpropagation

Let's do $dJ/dw_{11}^{(0)}$ as an example:

$$\frac{dJ}{dw_{11}^{(0)}} = \frac{dJ}{dz_1}\frac{dz_1}{dw_{11}^{(0)}} = \frac{dJ}{dz_1}\left(\frac{dz_1}{dh_1}\frac{dh_1}{dw_{11}^{(0)}}\right)$$



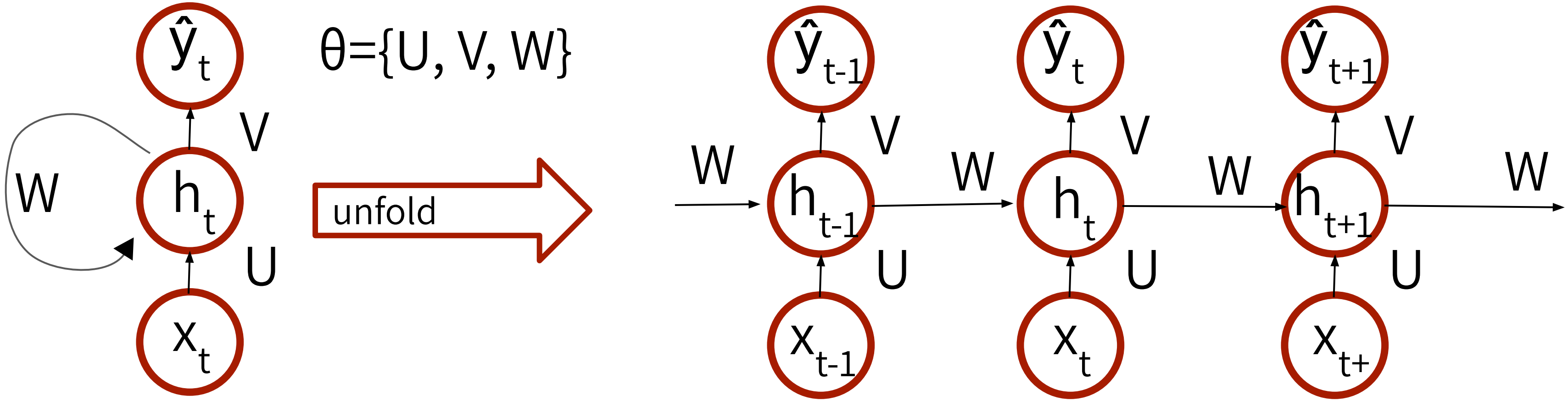
Use chain rule!

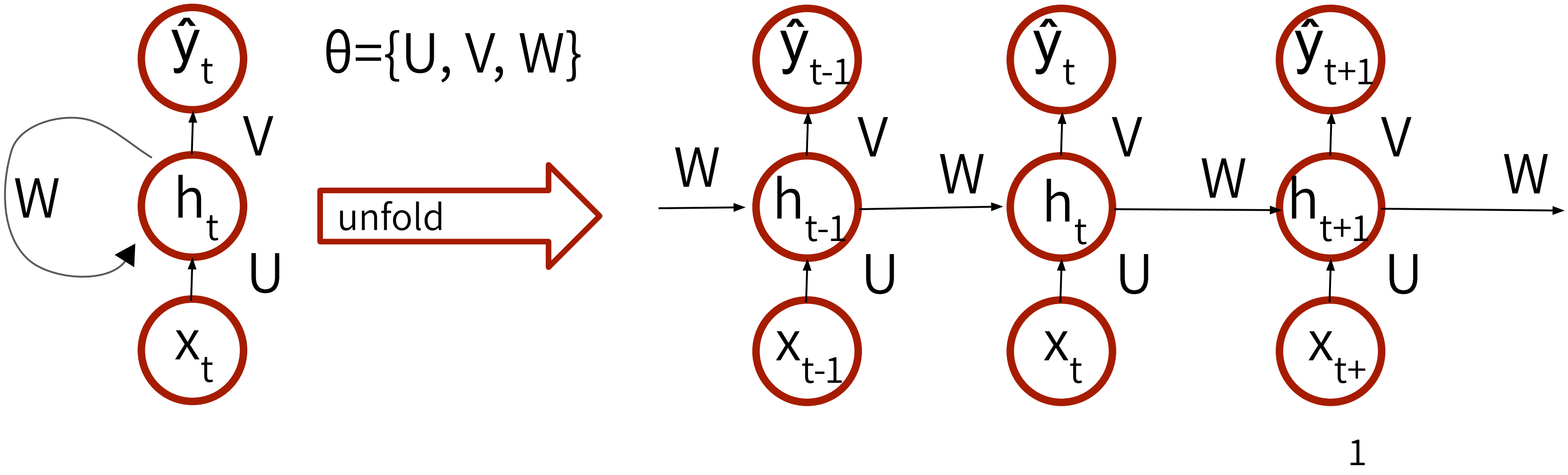
For a fixed objective J **and** a fixed architecture, you can compute a closed form for every derivative.

Recurrent NNs

Recurrent NNs

- Input is a sequence of tokens $(x_1, x_2, \ldots, x_T)$
- Output is a sequence of tokens $(y_1, y_2, \ldots, y_T)$
- Goal is map x_t to a "hidden state" h_t (a real-valued vector)
- Think of h_t is a nonlinear summary of $(x_1, \ldots, x_t)$
- Use h_{t-1} and x_t to predict y_t



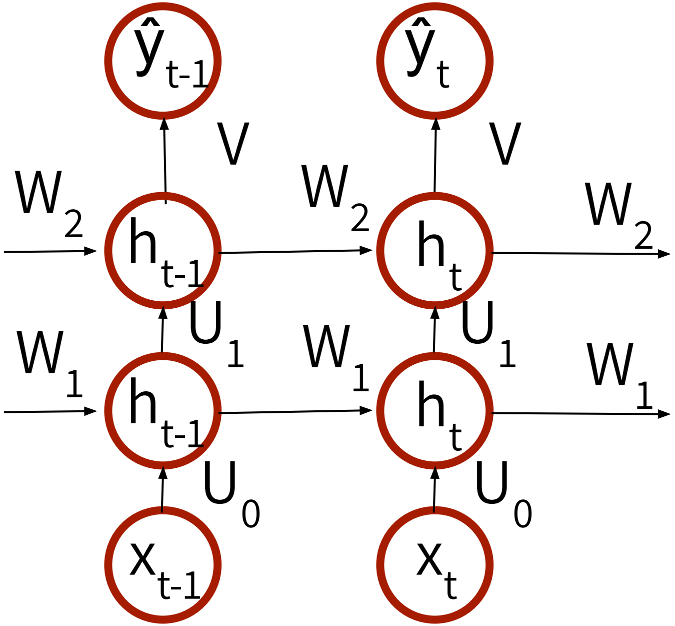


- U, V, W are <u>shared</u> over all timesteps (same ones!)
- The model does a forward pass for every single timestep: $f_\theta(x_t, h_{t-1})$
- <u>Objective</u> (assume x and y are discrete tokens)

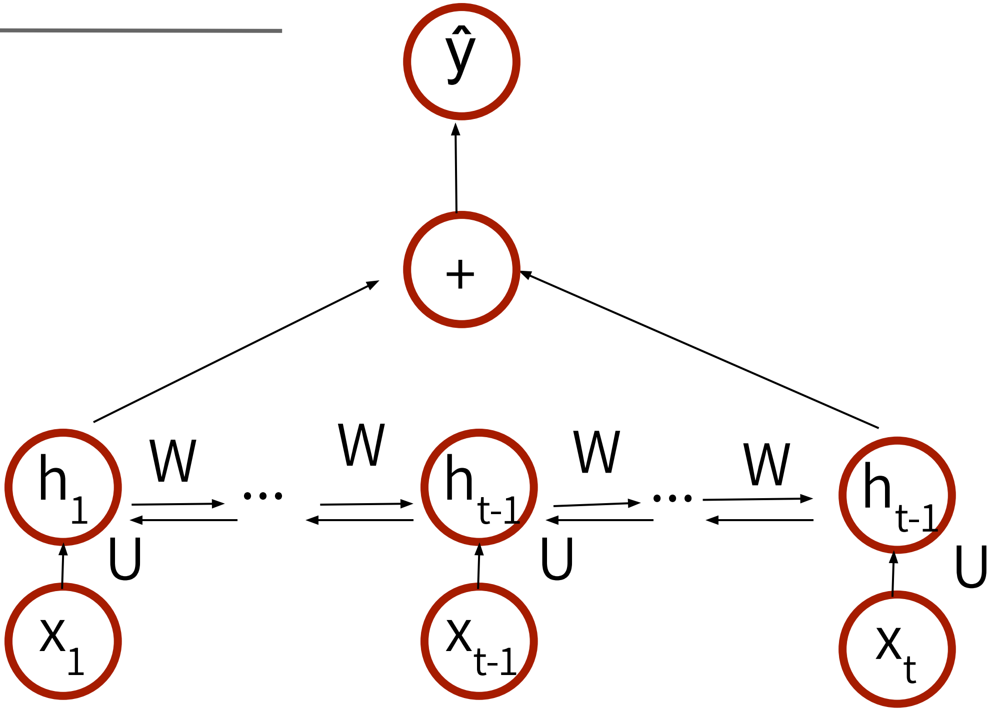
$$J(x,y,\theta) = -\sum_t \log p(y_t \mid x_1, \ldots, x_t) = -\sum_t \text{CrossEntropy}(y_t, f_\theta(x_t, h_{t-1}))$$

- Backpropagation through time (BPTT) [Rumelhart et. al. 1986, Werbos 1990]

Improvements to RNN

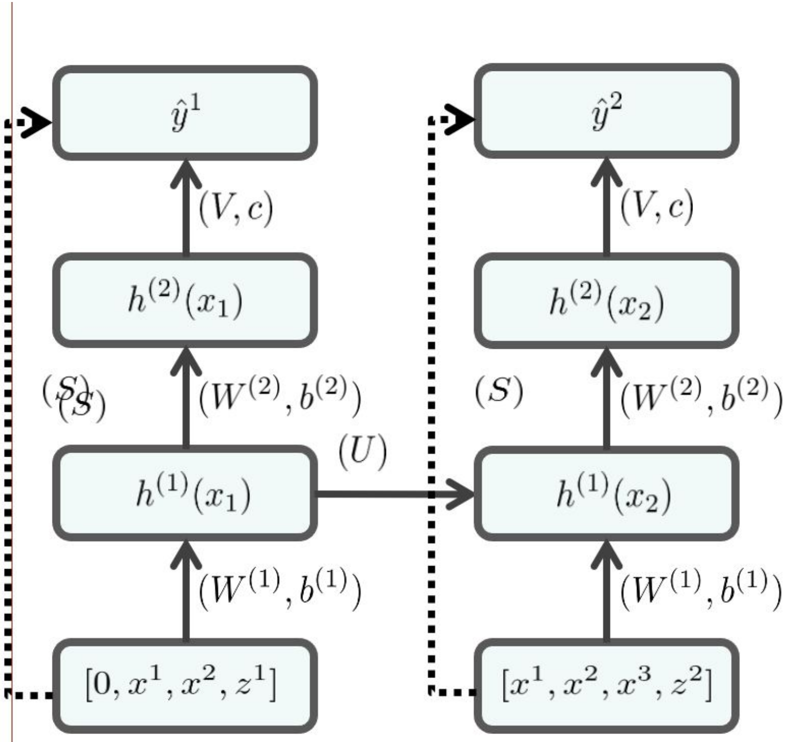


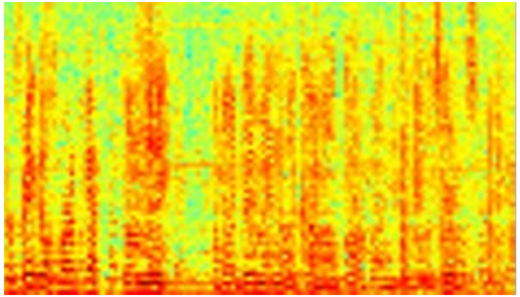
Stacked RNNs



Bi-directional RNNs

Recurrent networks for speech denoising





Output Layer

$$\hat{y}^2 = Vh^{(2)}(x_2) + Sx_2 + c$$

Hidden Layer

$$h^{(2)}(x_2) = \sigma(W^{(2)}h^{(1)}(x_2) + b^{(2)})$$

Hidden Layer

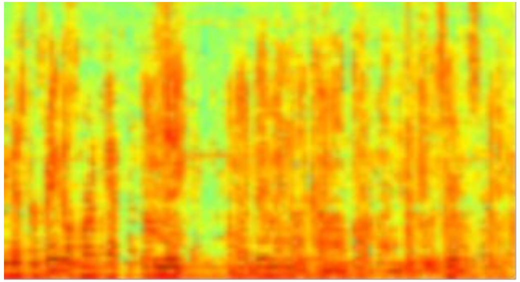
$$h^{(1)}(x_2) = \sigma(W^{(1)}x_2 + b^{(1)} + Uh^{(1)}(x_1))$$

Noisy Input

$\mathbf{X}$

Noise Estimate

$\mathbf{Z}$

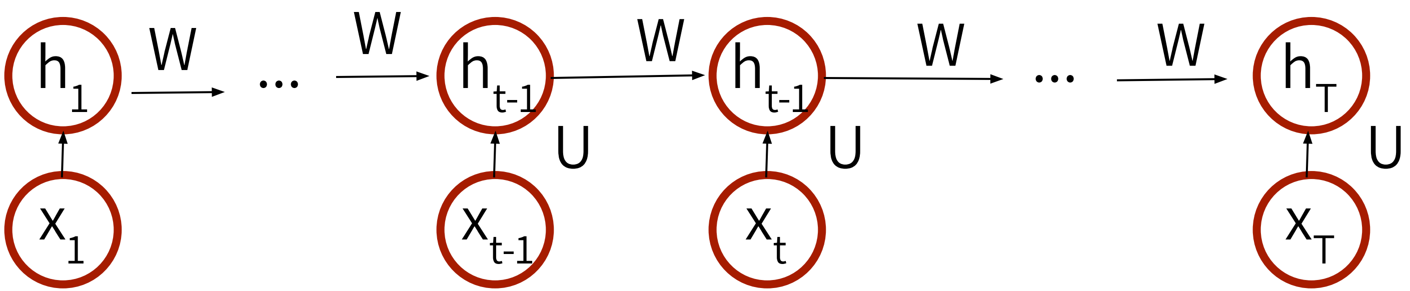


(Maas *et al*, 2013)

Encoder-Decoder Models

RNN Encoder

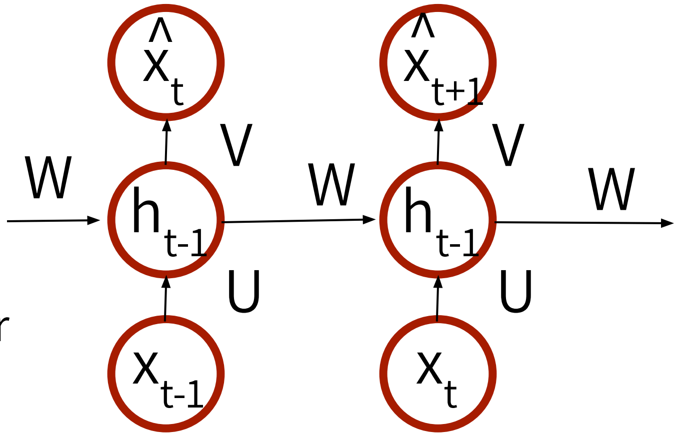
Input is a sequence of tokens $(x_1, x_2, \ldots, x_T)$. Goal is to summarize the sequence into a single vector.



RNN Decoder

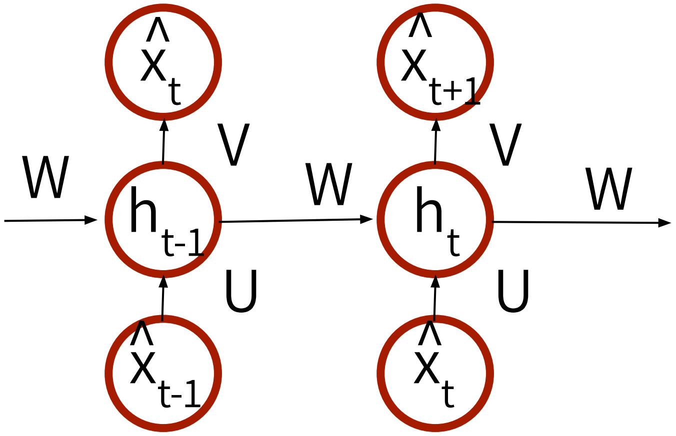
Input is a sequence of tokens $(x_1, x_{2,} \ldots, x_T)$, no output sequence. The model should learn $p(x_t \mid x_1, \ldots, x_{t-1})$.

“Teacher forcing”



Training

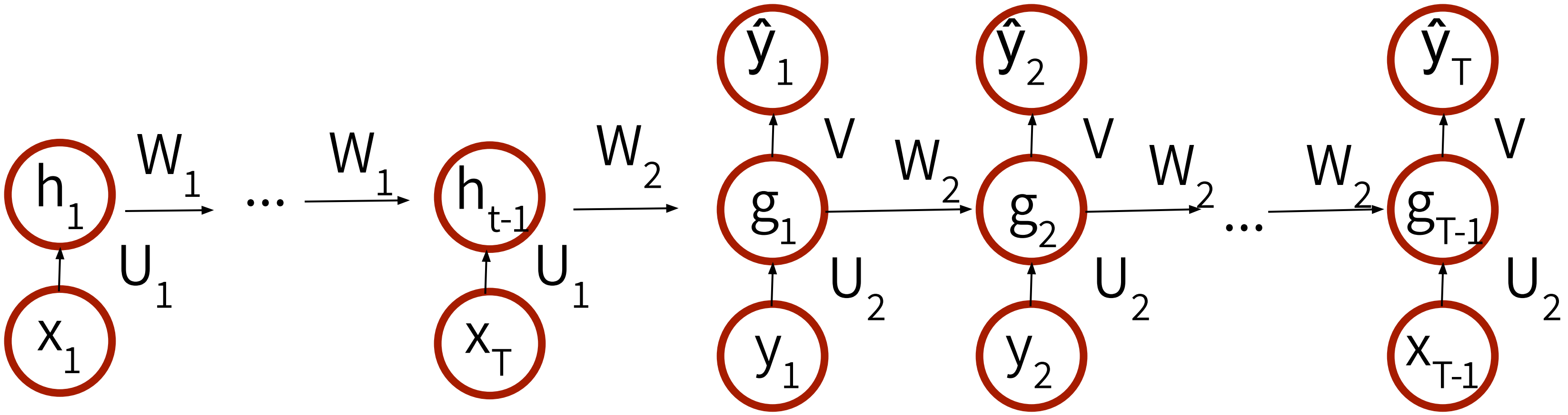
We don’t know the true x_t!

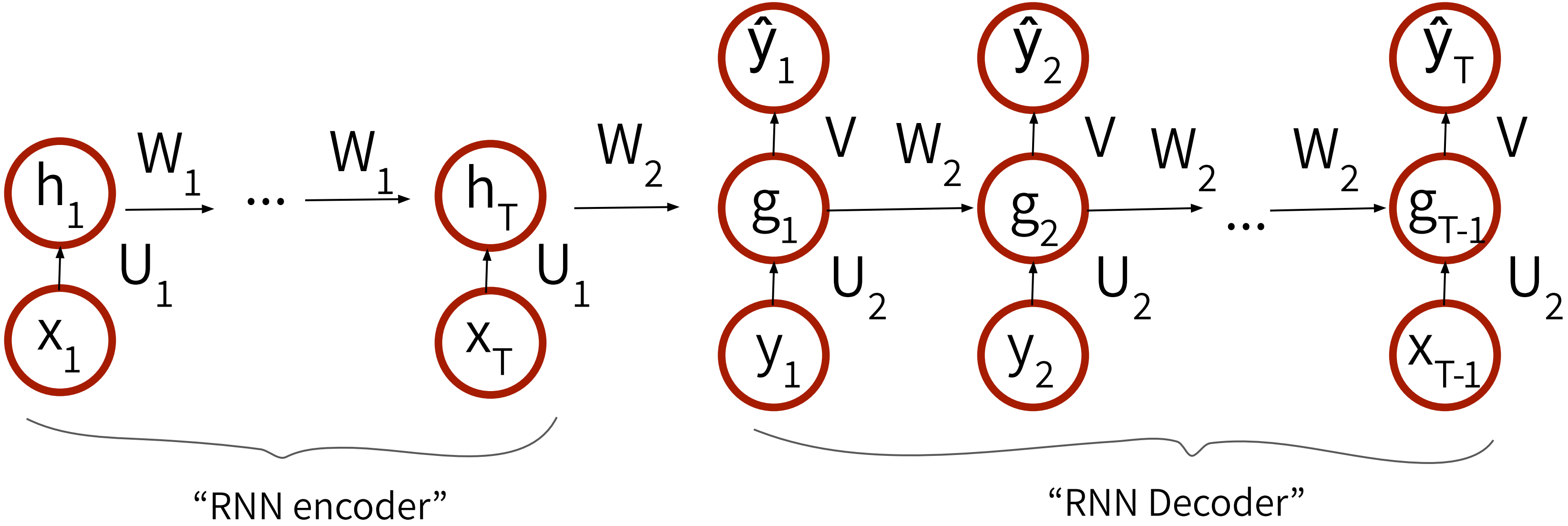


Generation

Seq2Seq [Cho et . al. 2014, Sustekever et. al. 2014]

- All that work and we have our first encoder-decoder model!
- Given an input and output sequence $(x_1, x_2, \dots, x_t)$, $(y_1, y_2, \dots, y_{t'})$, the model should capture $p(y_t \mid y_1, \dots, y_{t-1}, x_1, \dots, x_T)$. It gets to see all of x!





- Two different RNNs glue'd together (separate parameters)
- One of them encodes (x_1, …, x_T) into a summary vector, h_T
- The other one uses h_T to initialize a language model
- Train this just like an RNN language model (x = speaker 1, y = speaker 2)

Limitations of RNNs

- If you have a really long sequence (e.g. T=1000), hard to believe h_{854} will remember x_2
- h_t captures "local" info since it has to predict y_t (little incentive to remember h_{t-100}).

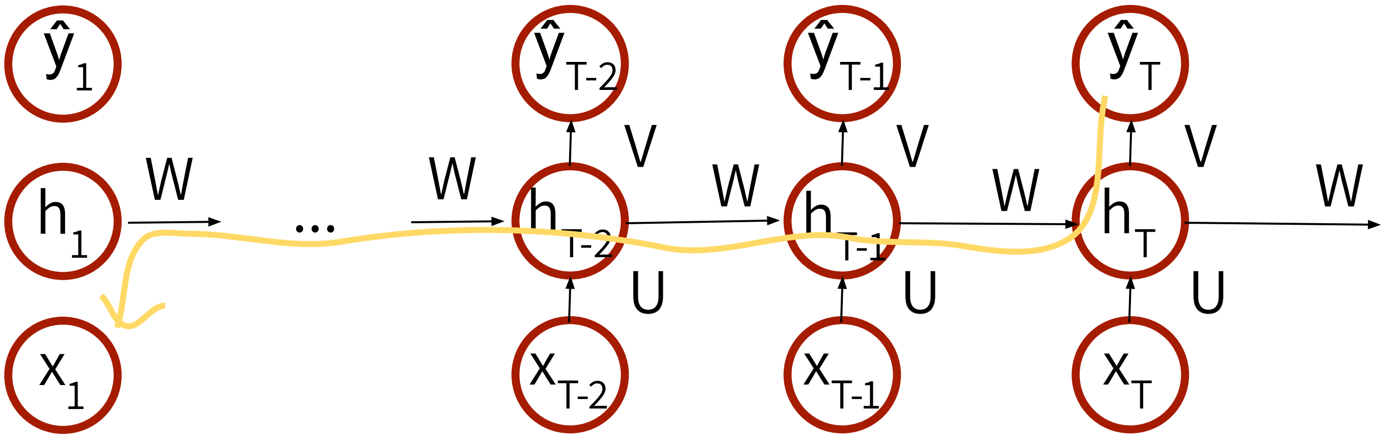
But long range dependencies are important.

Example: … the flights the airline was cancelling were full.

What flights??? What airline???

If you have building a chatbot, you might need to remember things from long ago.

Vanishing Gradients



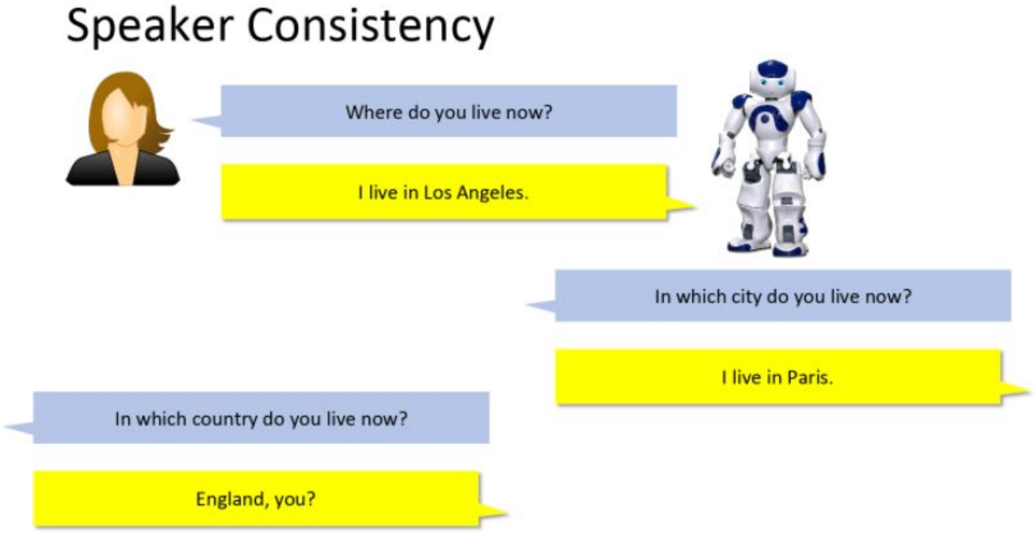
- Suppose T $\rightarrow \infty$. Say we want to calculate dy_t / dx_1
- What happens if $dh_t/dh_{t-1} < 1$ for all t?
- Impact of x_{t-s} on y_t decreases as s increases.

Attention

Attention [Bahdanau et. al. 2014]

Holy grail: capturing long term dependencies.

- Vanishing gradient problem & local dependency of RNNs.
- Attention gets at this more directly (and simply).

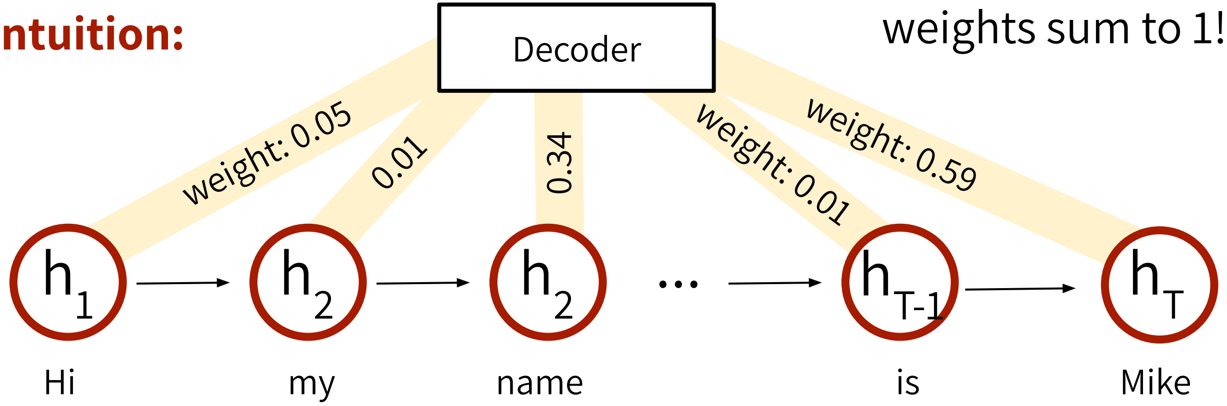


Attention [Bahdanau et. al. 2014]

Holy grail: capturing long term dependencies.

- Vanishing gradient problem & local dependency of RNNs.
- Attention gets at this more directly (and simply).

Intuition:



weights sum to 1!

Attention mechanism

$q \in R^d$: query ; $k_1, \ldots, k_T \in R^d$: keys ; $v_1, \ldots, v_T \in R^d$: values

Attention mechanism

$q \in R^d$: query ; $k_1, \ldots, k_T \in R^d$: keys ; $v_1, \ldots, v_T \in R^d$: values

Step 1: define a similarity function sim(q, k_t).

$\mathrm{sim}(q, k_t) = w_2 \mathrm{relu}(w_1[q; k_t] + b_1) + b_2$ [Bahdanau et. al. 2014] (MLP)

$\mathrm{sim}(q, k_t) = q^T W k_t$ [Luong et. al. 2015] (Bilinear)

$\mathrm{sim}(q, k_t) = q^T k_t$ [Luong et. al. 2015] (dot-pdt)

$\mathrm{sim}(q, k_t) = q^T k_t / \mathrm{sqrt}(d)$ [Luong et. al. 2015] (scaled dot-pdt)

Attention mechanism

$q \in R^d$: query ; $k_1, \ldots, k_T \in R^d$: keys ; $v_1, \ldots, v_T \in R^d$: values

Step 1: define a similarity function $\mathrm{sim}(q, k_t)$.

Step 2: compute attention weights a_t.

$$a_t = \frac{\exp\{\,\mathrm{sim}(q, k_t)\,\}}{\sum_{s=1}^{T} \exp\{\,\mathrm{sim}(q, k_s)\,\}}$$

Note $a_t \in [0, 1]$ for all t

Also $\sum_t a_t = 1$

Attention mechanism

$q \in R^d$: query ; $k_1, \ldots, k_T \in R^d$: keys ; $v_1, \ldots, v_T \in R^d$: values

Step 1: define a similarity function $\mathrm{sim}(q, k_t)$.

Step 2: compute attention weights a_t.

Step 3: attend to values vectors.

$c = \sum_{t=1}^{T} a_t v_t$ weighted linear combo of values!

Multi-headed Attention [Vaswani et. al. 2017]

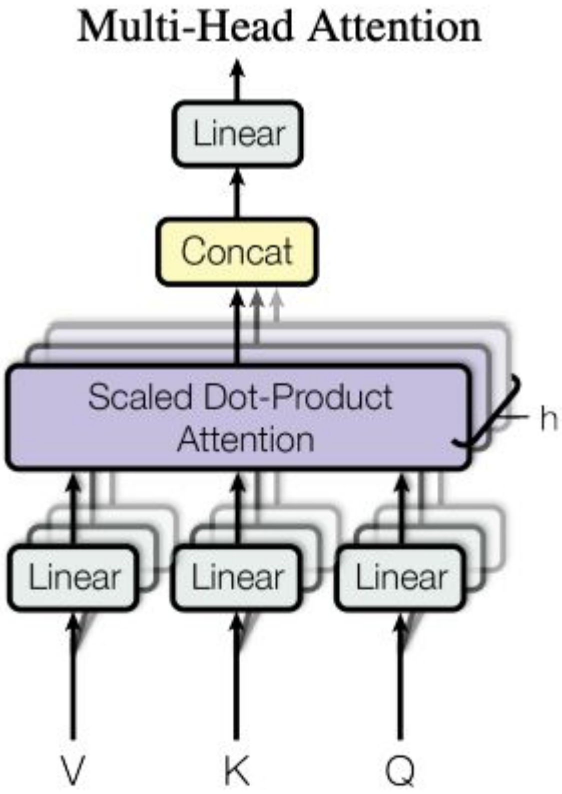
What if I want to pay attention to different things at the same time!?

This is my big red dog, Clifford.

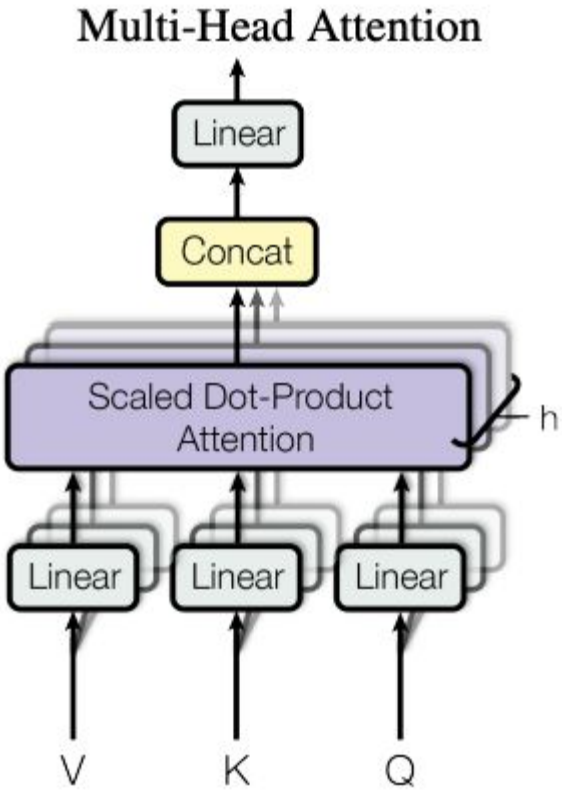
Content-based	This is my big red dog, Clifford.
Description-based	This is my big red dog, Clifford.
Reference-based	This is my big red dog, Clifford.

What's useful depends on the task. How do I pick what to do?

<u>Idea</u> [Vaswani et. al. 2017]: Don't pick. Pay attention as if you had "multiple heads".



Idea [Vaswani et. al. 2017]: Don't pick. Pay attention as if you had "multiple heads".

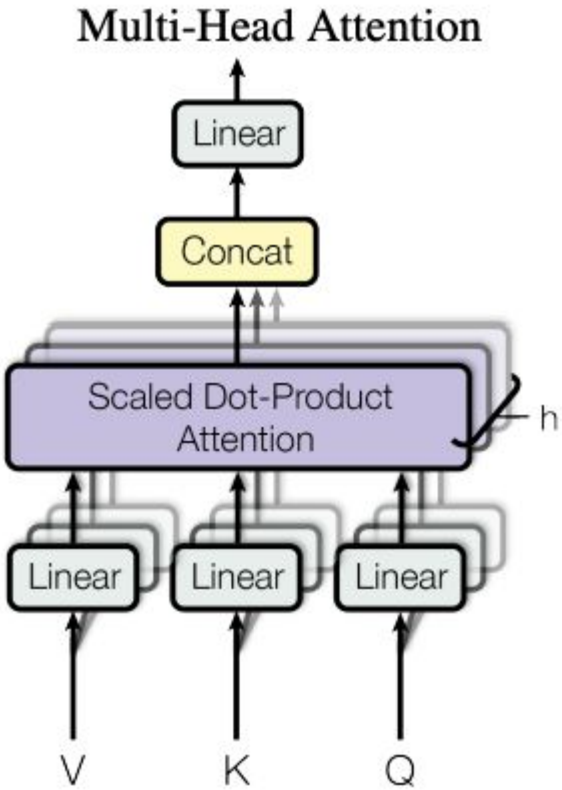


Pick H heads.

For h in range(H):

$q^{(h)} = MLP_h(q)$
$k^{(h)}_t = MLP_h(k_t)$ for all t
$v^{(h)}_t = MLP_h(v_t)$ for all t

Idea [Vaswani et. al. 2017]: Don't pick. Pay attention as if you had "multiple heads".



Pick H heads.

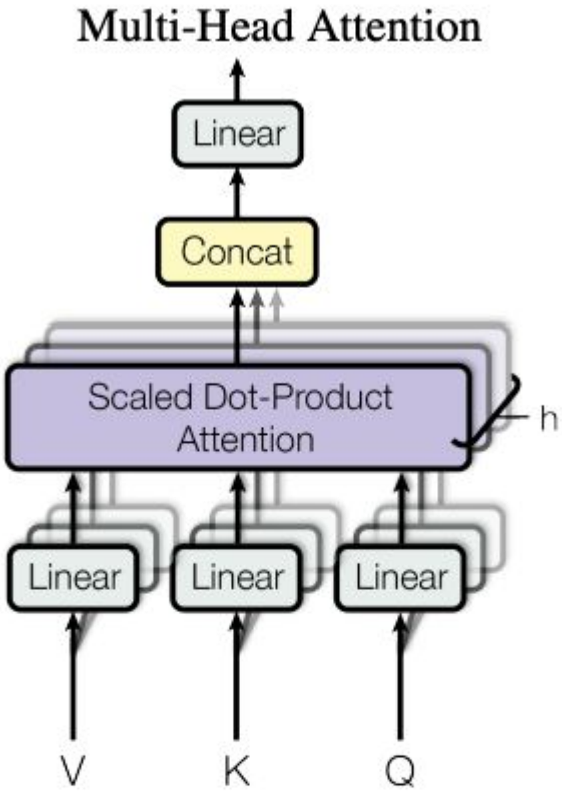
For h in range(H):

$q^{(h)} = MLP_h(q)$
$k^{(h)}_t = MLP_h(k_t)$ for all t
$v^{(h)}_t = MLP_h(v_t)$ for all t

$$a^{(h)}_t = \frac{\exp\{ q^{(h)T} k_t^{(h)} / \mathrm{sqrt}(d) \}}{\sum^T_{s=1} \exp\{ q^{(h)T} k_s^{(h)} / \mathrm{sqrt}(d) \}}$$

$$c^{(h)} = \sum^T_{t=1} a^{(h)}_t v^{(h)}_t$$

Idea [Vaswani et. al. 2017]: Don’t pick. Pay attention as if you had “multiple heads”.



Pick H heads.

For h in range(H):

$q^{(h)} = MLP_h(q)$
$k^{(h)}_t = MLP_h(k_t)$ for all t
$v^{(h)}_t = MLP_h(v_t)$ for all t

$$a^{(h)}_t = \frac{\exp\{ q^{(h)T} k_t^{(h)} / \mathrm{sqrt}(d) \}}{\sum^T_{s=1} \exp\{ q^{(h)T} k_s^{(h)} / \mathrm{sqrt}(d) \}}$$

$c^{(h)} = \sum^T_{t=1} a^{(h)}_t v^{(h)}_t$

$c_{all} = \mathrm{concat}(c^{(1)}, \ldots, c^{(H)})$
$c = \mathrm{linear}(c_{all})$ # project to smaller dimension

Self Attention [Vaswani et. al. 2017]

Simple idea: query, keys and values in attention are the same.

Take some sequence $(x_1, x_2, \ldots, x_T)$. For every t, build:

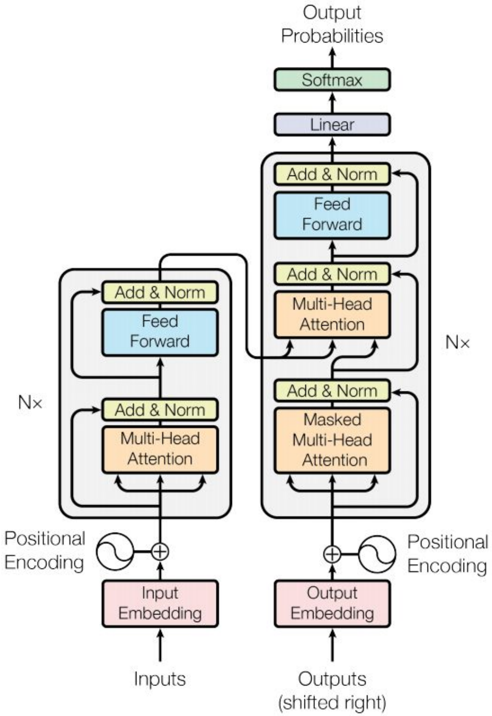
$$q_t = \mathrm{MLP}(x_t) \;,\; k_t = \mathrm{MLP}(x_t) \;,\; v_t = \mathrm{MLP}(x_t)$$

Then we can extract a sequence:

$$c_t = \sum^{T}_{t=1} a_t\, v_t \;\Rightarrow\; (c_1, c_2, \ldots, c_T)$$

Transformers (SOTA)

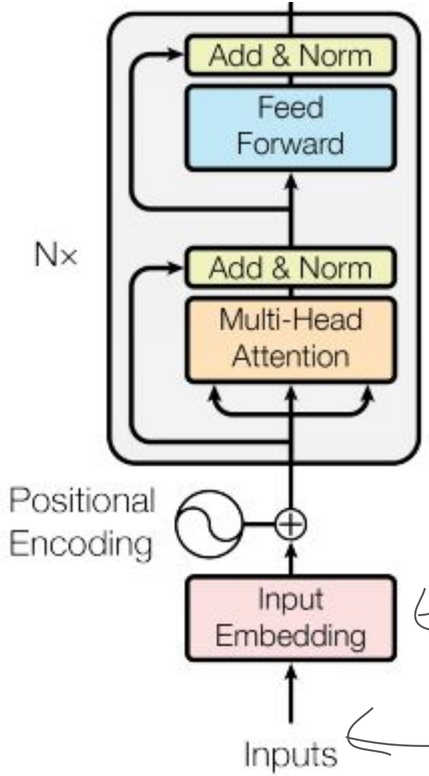
Transformer: Self-attention Nets



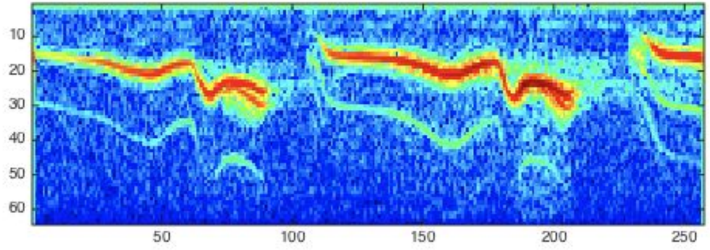
- A transformer layer is composed of an <u>encoder</u> and a <u>decoder</u>.
- Both use the same building blocks.

Similar to RNNs, here...

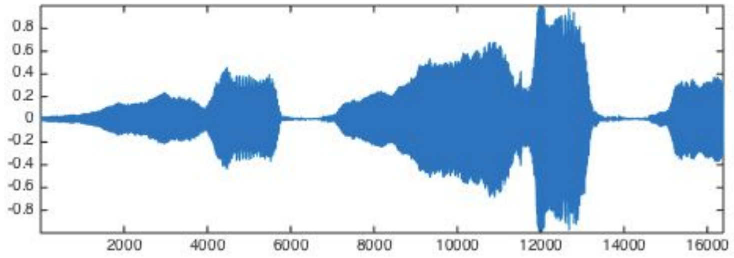
- Encoder sees $(x_1, x_2, \ldots, x_T)$ and outputs a hidden sequence $(h_1, h_2, \ldots, h_T)$.
- Decoder sees $(x_1, x_2, \ldots, x_{T-1})$ and outputs predictions for $(x_2, x_3, \ldots, x_T)$.



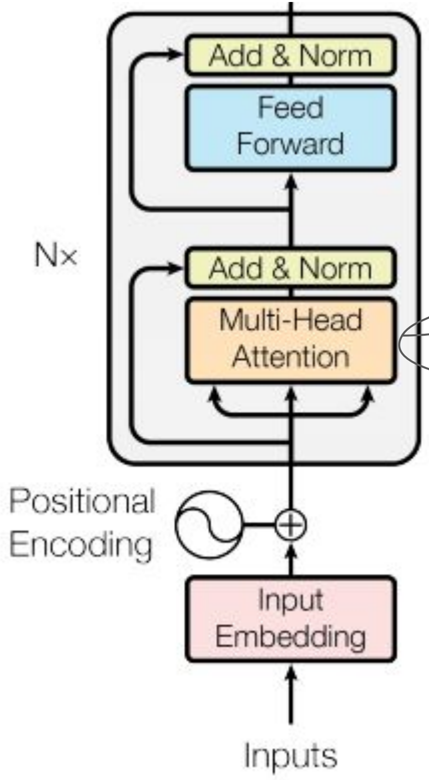
Transformer encoder



$(e_1, e_2, \ldots, e_T)$ e.g. mel spectrogram



$(x_1, x_2, \ldots, x_T)$ e.g. a waveform

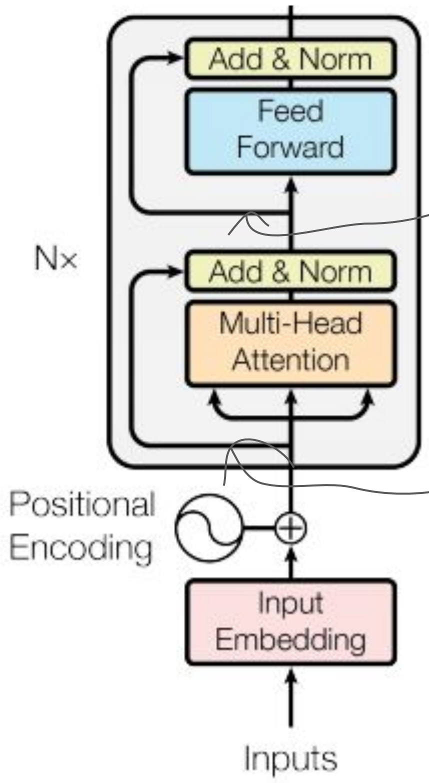


Transformer encoder

We've seen this! We do self-attention with H heads on the lookup embeddings.

<u>Inputs</u> $(e_1, e_2, \ldots, e_T)$, each e_t is now a vector!

<u>Outputs</u> $(c_1, c_2, \ldots, c_T)$ each $c_t \in R^d$



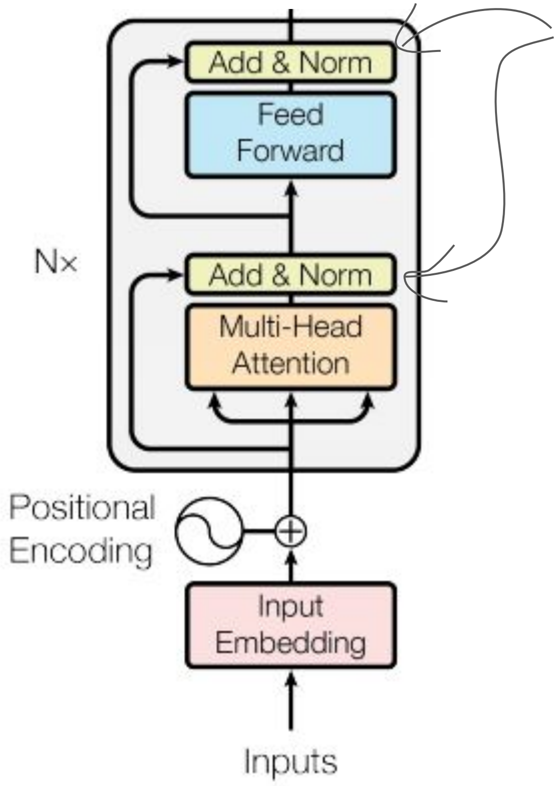
Transformer encoder

Residual sum

$(e_1, e_2, \ldots, e_T)$ inputs (spectrogram features)

$(c_1, c_2, \ldots, c_T)$ attention outputs

$h_t^{res} = c_t + e_t$ residual output



Transformer encoder

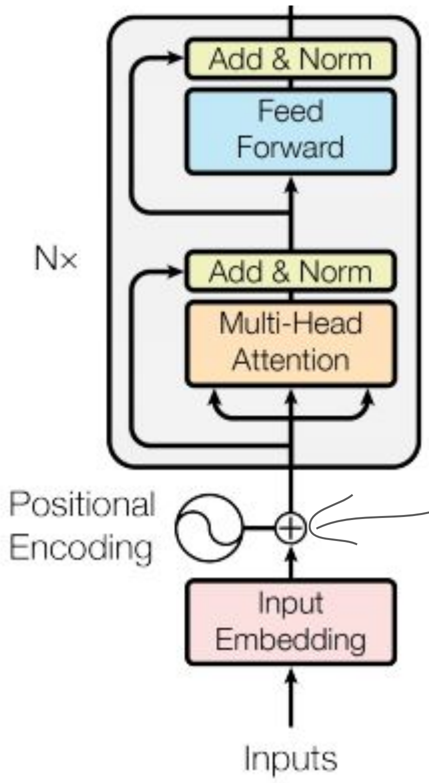
Layer normalization

$$h^{norm} = \frac{h^{res} - E[h^{res}]}{\mathrm{sqrt}(Var[h^{res}] + \varepsilon)} * \gamma + \beta$$

Note $\{\gamma, \beta\} \subseteq \theta$ e.g. learnable parameters.

$$h^{res} = (h_1^{res}, h_2^{res}, \ldots, h_T^{res}).$$

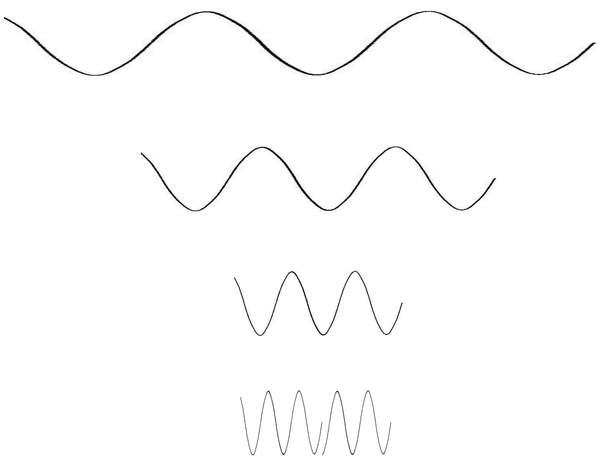
- The mean and variance are over the sequence of size T.
- Not like batch norm (which is over a batch of examples). This is only on 1 example.



Transformer encoder

- Unlike RNNs, transformers have no order!
- But speech is left-to-right so it might be useful to tell the model that.

Positional Encodings



Input: $(x_1, x_2, x_3, \ldots, x_T)$
Position: $(1, 2, 3, \ldots, T)$

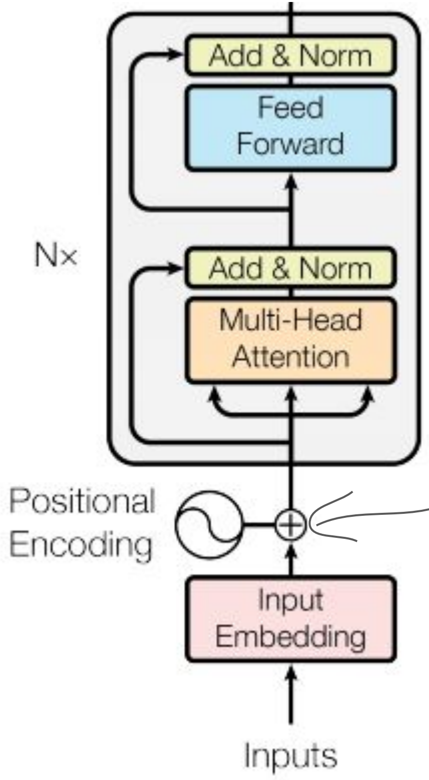
But we can be a bit more clever:

$PE(t, 2i) = \sin(t/10000^{2i/d})$
$PE(t, 2i{+}1) = \cos(t/10000^{2i/d})$

$PE(t) = [PE(t, 0), PE(t, 1), \ldots, PE(t, d)]$

Add embedding of t-th token $e_t = e_t + PE(t)$.

Positional Encodings



Transformer encoder

Input: $(x_1, x_2, x_3, \ldots, x_T)$
Position: $(1, 2, 3, \ldots, T)$

But we can be a bit more clever:

$PE(t, 2i) = \sin(t/10000^{2i/d})$
$PE(t, 2i+1) = \cos(t/10000^{2i/d})$

$PE(t) = [PE(t, 0), PE(t, 1), \ldots, PE(t, d)]$

Add embedding of t-th token $e_t = e_t + PE(t)$.

- Assigns every timestep a unique waveform
- No need to specify maximum length

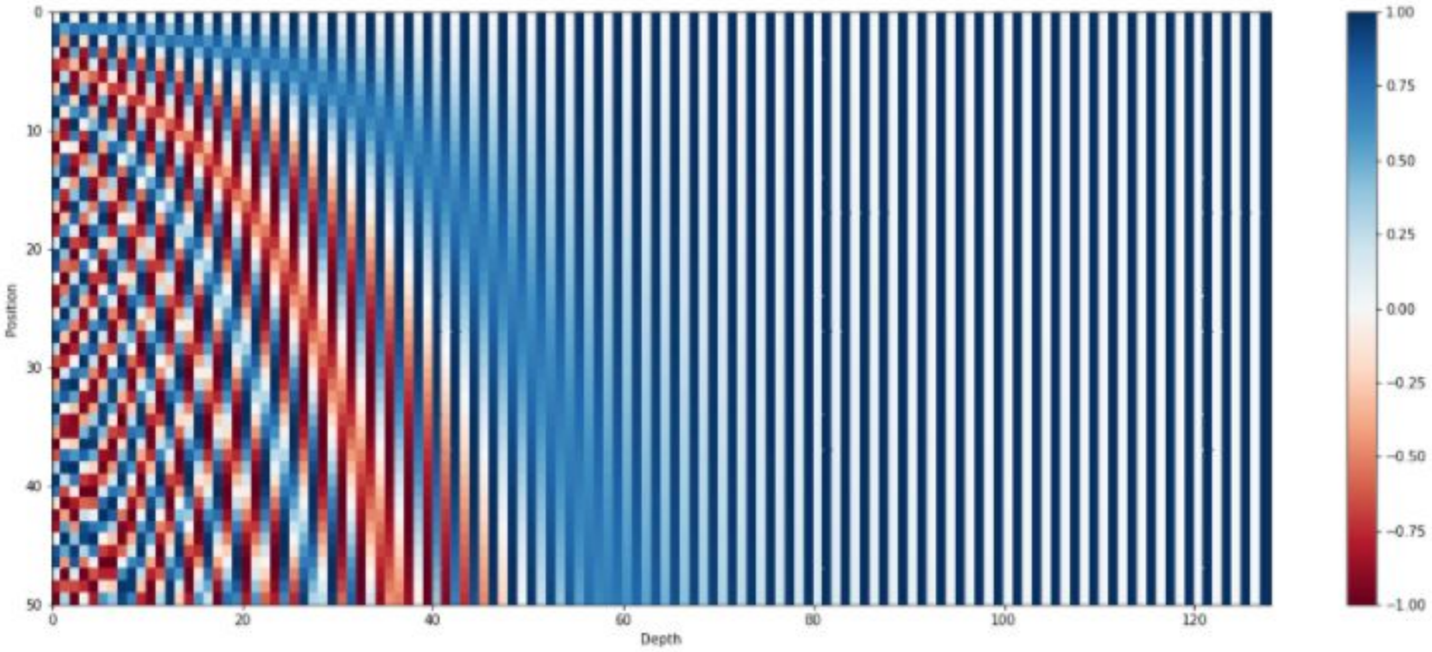
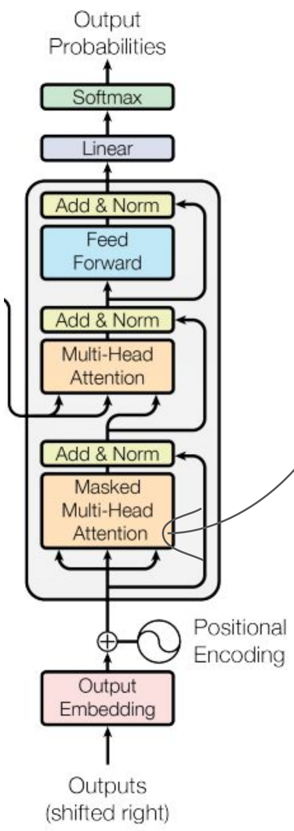


Figure 2 – The 128-dimensional positonal encoding for a sentence with the maximum lenght of 50. Each row represents the embedding vector $\overrightarrow{p_t}$

[Source post]



Transformer decoder

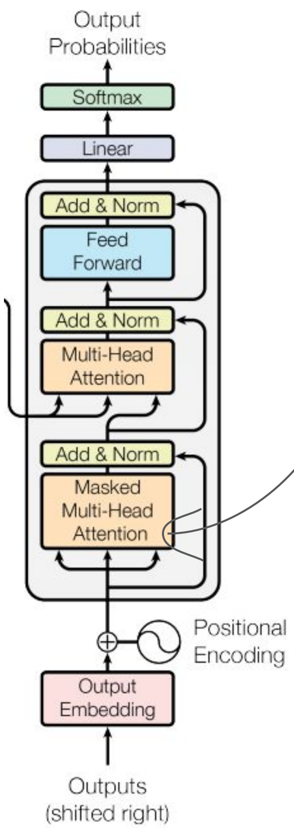
Masked Multi-head Attention:

We can’t do exactly what we do in the encoder b/c we don’t want to bleed future info.

$$(x_1, x_2, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_{T-1}, x_T)$$

current

Cheating if we see this b/c in test time, we don’t have access to > t+1

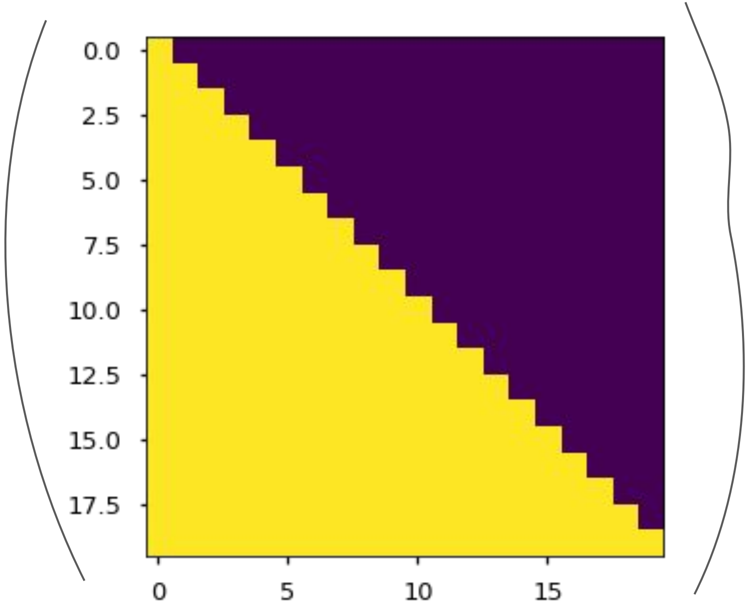


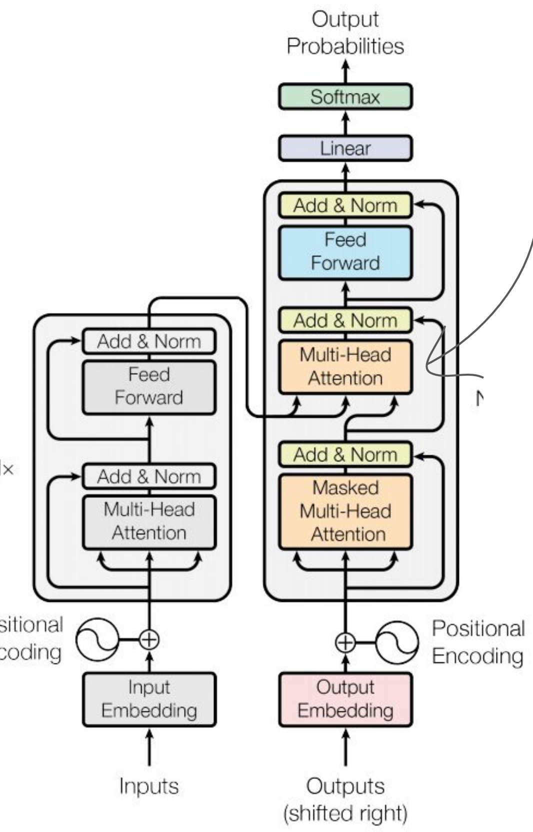
Transformer decoder

Masked Multi-head Attention:

Recall, q : query ; $k_1, \ldots, k_T$: keys ; $v_1, \ldots, v_T$: values

$$\text{maskedsim}(q, k_t, m) = m^T(q^T k_t) \,/\, \text{sqrt}(d)$$

$m =$ 



Transformer decoder

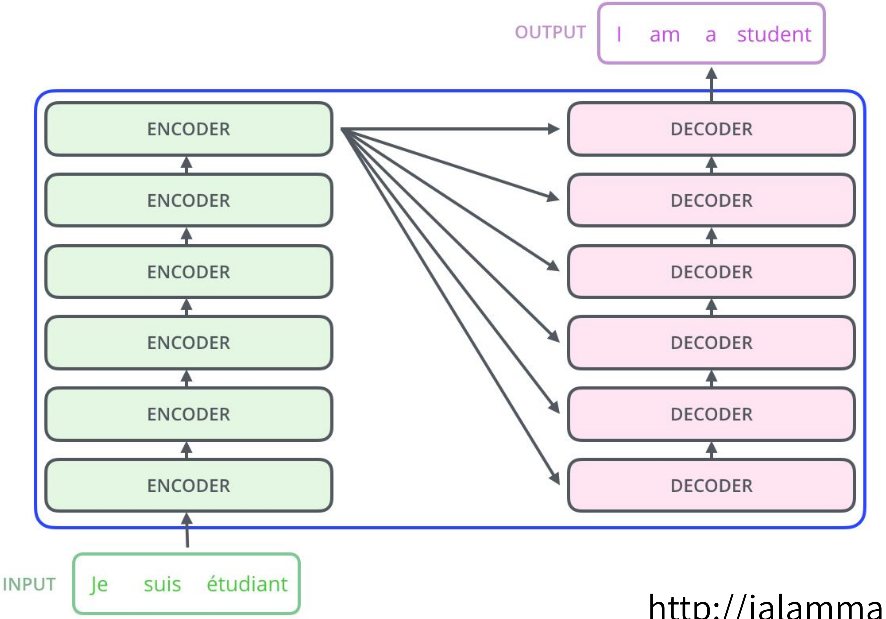
Encoder Multi-head Attention:

- Output of encoder: $(h_1^{enc}, h_2^{enc}, \ldots, h_T^{enc})$.
- Use this for keys and values in attention.
- Query vectors come from decoder.

- This blends information from encoder into the decoder.
- Note: no bleeding problem here!
- SOTA speech recognition network is a variant of this idea for online+offline training with weight sharing

Stacked Transformers

The trend is make things deep. A single transformer encoder or decoder returns a sequence of the same signature as the input.



http://jalammar.github.io/illustrated-transformer

Resources

Neural network basics: http://cs229.stanford.edu, https://www.deeplearningbook.org

RNNs: http://web.stanford.edu/class/cs224n/index.html#schedule , Sequence to sequence learning with neural networks [Sutskever et. al. 2014]

Transformers: Attention is all you need [Vaswani et. al. 2017], http://jalammar.github.io/illustrated-transformer

Code: https://huggingface.co/transformers

CS224N deep learning tutorial videos (linked on course page)

Starting thinking about project ideas!

Appendix

Neural Chatbots

How do you train a neural network to chat?

Neural Chatbots

How do you train a neural network to chat?

- Handwritten rules? (Elizabot)... interesting but wouldn't pass turing
- Finite state machines?... good for some tasks but too limited

Neural Chatbots

How do you train a neural network to chat?

- Handwritten rules? (Elizabot)... interesting but wouldn't pass turing
- Finite state machines?... good for some tasks but too limited
- Treat it like a translation problem! End-to end neural networks.

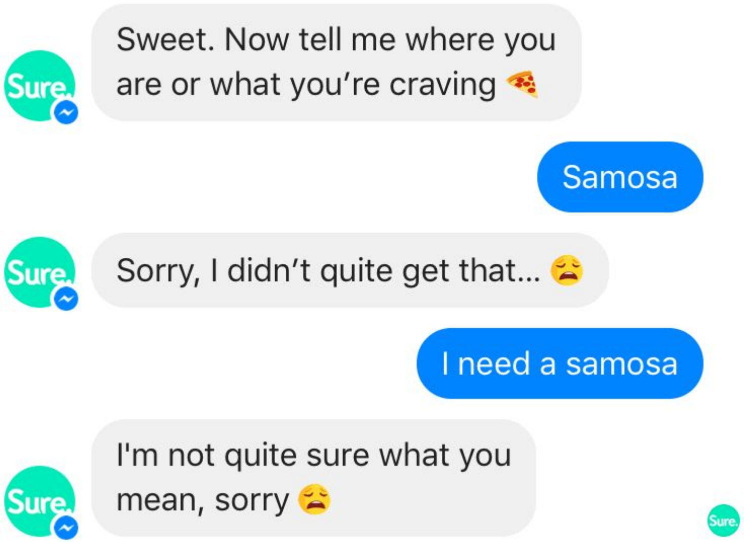
Machine Translation: Hi how are you? → Hola! Cómo estás?

Chatbot: Hi how are you? → Not bad, you?

Challenges of Chatbots

Even the best encoder decoder model doesn’t “solve” chat bots.

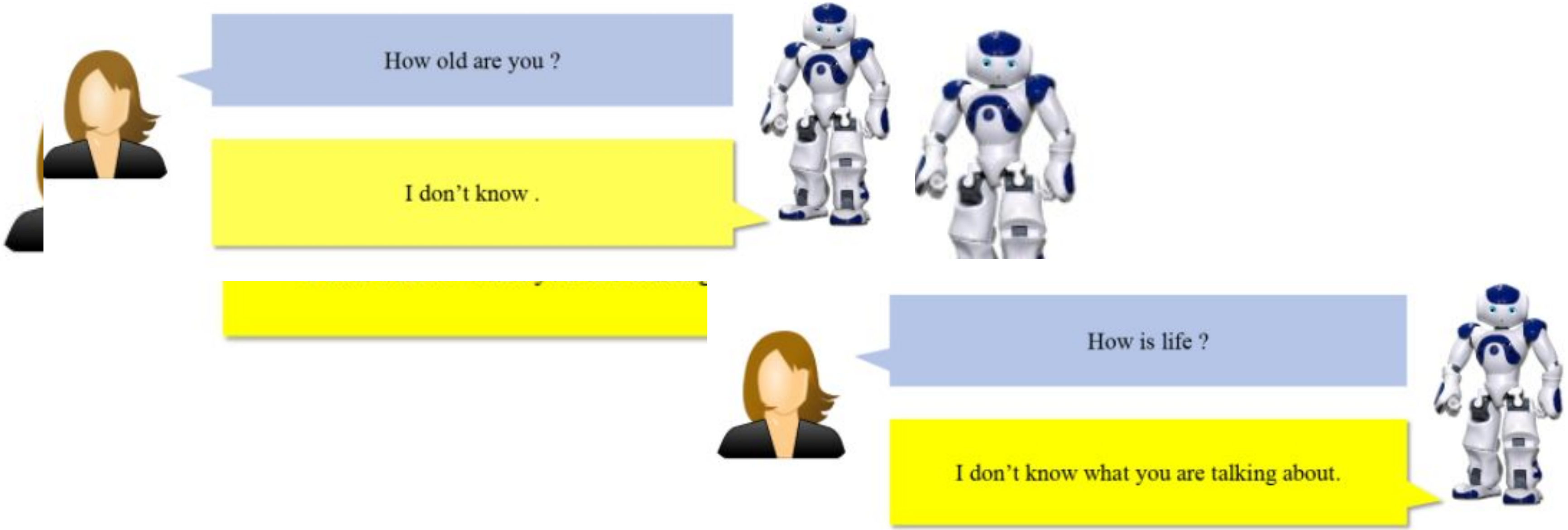
Here are 2 examples of failure modes and possible improvements.



Thomas Gouritin, 2017, TheNextWeb.com

Generic Responses

Problem: Easy for the model to say something in domain but generic in response to everything e.g. "I don't know" [Sordoni et. al. 2015]



Generic Responses

<u>Problem:</u> Easy for the model to say something in domain but generic in response to everything e.g. “I don’t know” [Sordoni et. al. 2015]

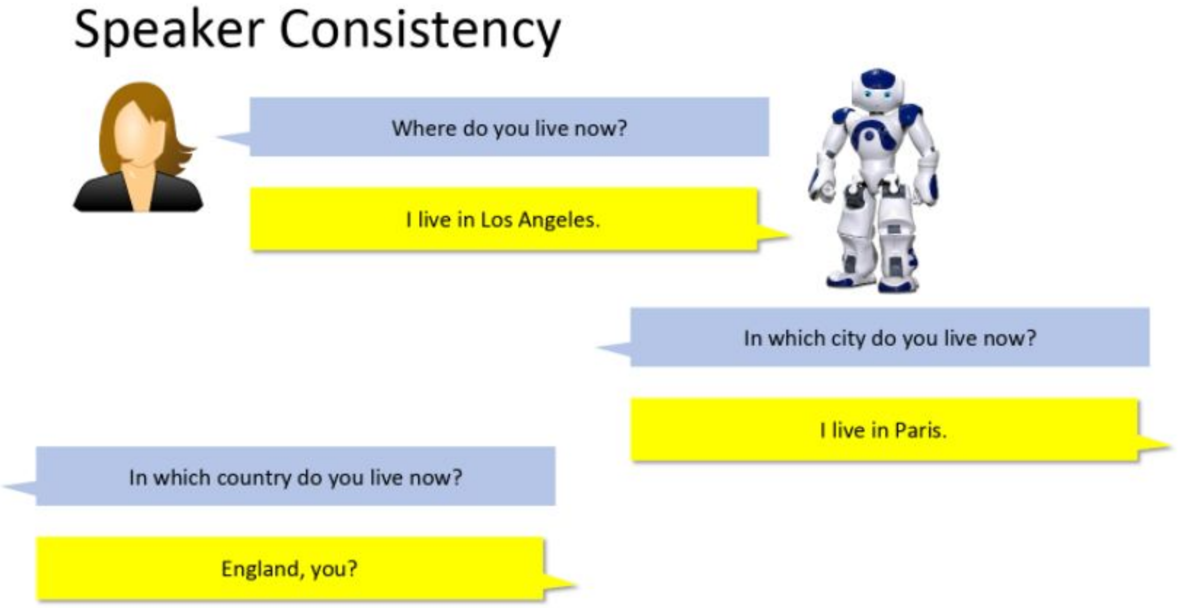
<u>A Solution:</u> Auxiliary objectives!

Optimize for high mutual information between source and response!

$$J = \underbrace{-\log p(\text{“I don’t know”} \mid \text{“how’s life”})}_{\text{Regular objective}} + \underbrace{\overset{\sim 0}{MI(\text{“i don’t know , “how’s life”})}}_{\text{Regularization}}$$

Speaker Consistency

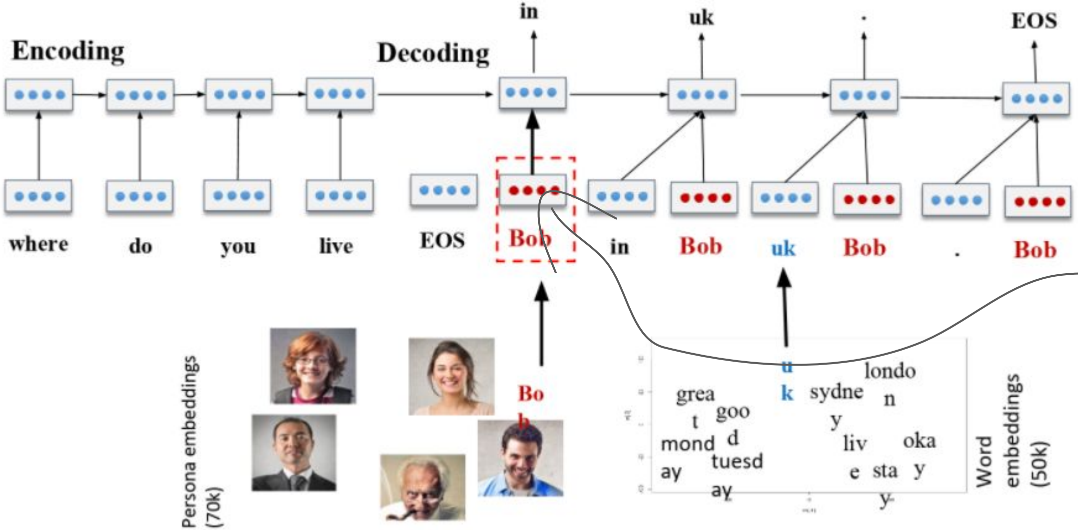
Problem: Just English sounding responses isn't enough. Chatbots should not contradict themselves factually.



Speaker Consistency

Problem: Just English sounding responses isn't enough. Chatbots should not contradict themselves factually.

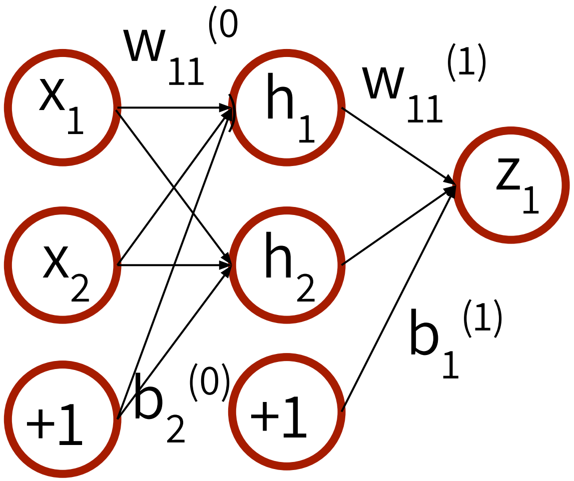
A Solution: Auxiliary information! Remember who you are talking to!



Can always add more info.

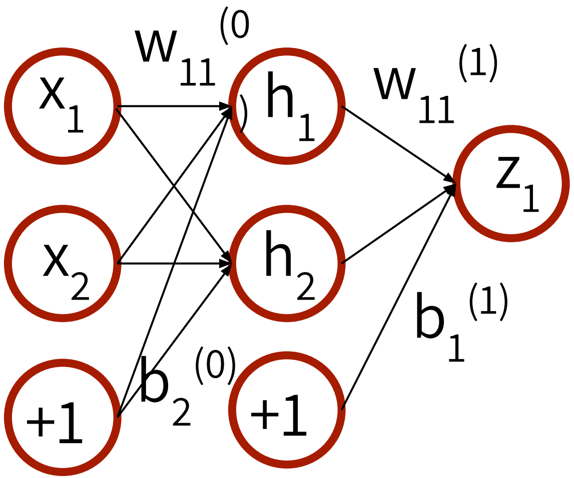
Backpropagation

Do a simple example with a small MLP.



Compute $\nabla_\theta L(x, y, \theta)$ of objective wrt parameters.

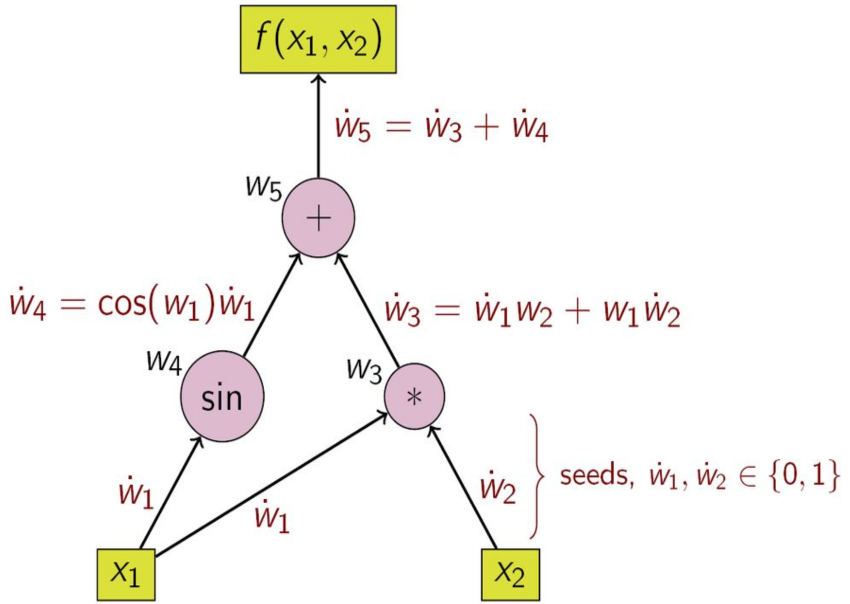
$$\nabla_\theta J(x, y, \theta) = \begin{bmatrix} dJ/dw_{11}^{(0)} & dJ/dw_{11}^{(1)} \\ dJ/dw_{12}^{(0)} & dJ/dw_{21}^{(1)} \\ dJ/db_{1}^{(0)} & dJ/db_{1}^{(1)} \\ dJ/dw_{21}^{(0)} & \\ dJ/dw_{22}^{(0)} & \\ dJ/db_{2}^{(0)} & \end{bmatrix}$$



Autodifferentiation

- Deriving these by hand is annoying.
- If you have new objective functions, this could be really intractable.
- **Idea:** if you manually specify the derivative for a set of “basic” operations, you can calculate derivative of complicated functions using chain rule.

Rall, Louis B., and RALL LB. "Automatic differentiation: Techniques and applications." (1981).



Create new independent variables in forward pass:

$z = f(x_1, x_2)$
$= x_1 x_2 + \sin x_1$
$= w_1 w_2 + \sin w_1$
$= w_3 + w_4$
$= w_5$

Operations to compute value	Operations to compute derivative
$w_1 = x_1$	$\dot{w}_1 = 1$ (seed)
$w_2 = x_2$	$\dot{w}_2 = 0$ (seed)
$w_3 = w_1 \cdot w_2$	$\dot{w}_3 = w_2 \cdot \dot{w}_1 + w_1 \cdot \dot{w}_2$
$w_4 = \sin w_1$	$\dot{w}_4 = \cos w_1 \cdot \dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$

$\dot{w}_1 = dw_1/dx_1$
want df/dx_1 so w_1=1

product rule

chain rule + sine rule

https://en.wikipedia.org/wiki/Automatic_differentiation

Magic :) 🔮

```
model.train()
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    output = model(data)
    loss = F.nll_loss(output, target)
    loss.backward()
    optimizer.step()
    if batch_idx % args.log_interval == 0:
        print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(train_loader.dataset),
            100. * batch_idx / len(train_loader), loss.item()))
        if args.dry_run:
            break
```

https://github.com/pytorch/examples/blob/master/mnist/main.py

Long Short Term Memory

LSTMs have the ability to "forget" information and "store" information that could be useful later [Schmidhuber 1997].

Forget gate:	$f_t = \sigma(U_f h_{t-1} + W_f x_t)$	Pick what to "forget"
	$k_t = c_{t-1} * f_t$	Do the "forgetting"
Add gate:	$g_t = \sigma(U_g h_{t-1} + W_g x_t)$	Usual RNN function
	$I_t = \sigma(U_i h_{t-1} + W_i x_t)$	Pick what to "add"
	$j_t = g_t * i_t$	Do the "adding"
	$c_t = j_t + k_t$	context is some of last context and some new stuff

Forget gate:	$f_t = \sigma(U_f h_{t-1} + W_f x_t)$	Pick what to "forget"
	$k_t = c_{t-1} * f_t$	Do the "forgetting"
Add gate:	$g_t = \sigma(U_g h_{t-1} + W_g x_t)$	Usual RNN function
	$I_t = \sigma(U_i h_{t-1} + W_i x_t)$	Pick what to "add"
	$j_t = g_t * i_t$	Do the "adding"
	$c_t = j_t + k_t$	context is some of last context and some new stuff
Output gate:	$o_t = \sigma(U_o h_{t-1} + W_o x_t)$	Pick what to use for current timestep
	$H_t = o_t * \tanh(c_t)$	Do the partitioning

Forget gate: $f_t = \sigma(U_f h_{t-1} + W_f x_t)$ Pick what to "forget"

$k_t = c_{t-1} * f_t$ Do the "forgetting"

Add gate: $g_t = \sigma(U_g h_{t-1} + W_g x_t)$ Usual RNN function

$I_t = \sigma(U_i h_{t-1} + W_i x_t)$ Pick what to "add"

$j_t = g_t * i_t$ Do the "adding"

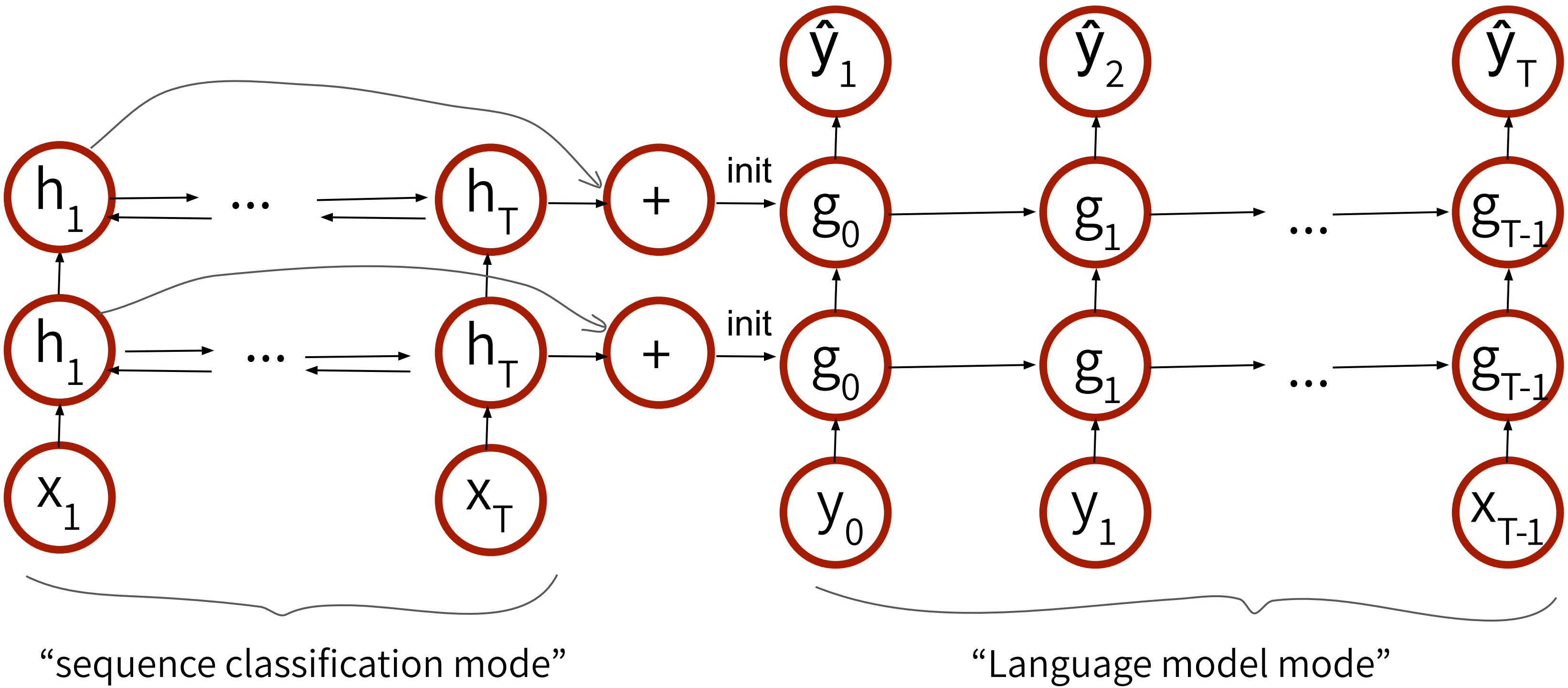
$c_t = j_t + k_t$ context is some of last context and some new stuff

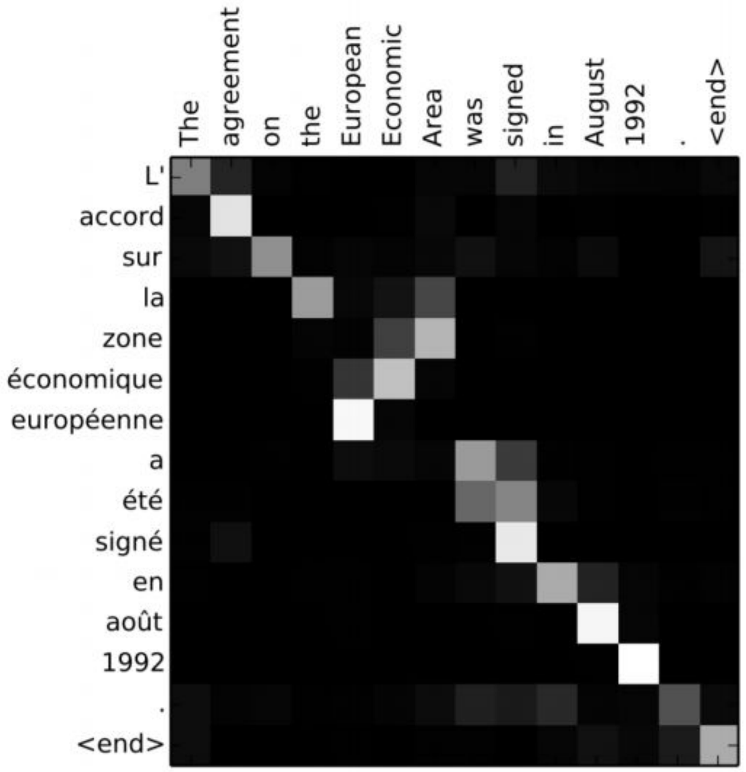
Output gate: $o_t = \sigma(U_o h_{t-1} + W_o x_t)$ Pick what to use for current timestep

$H_t = o_t * \tanh(c_t)$ Do the partitioning

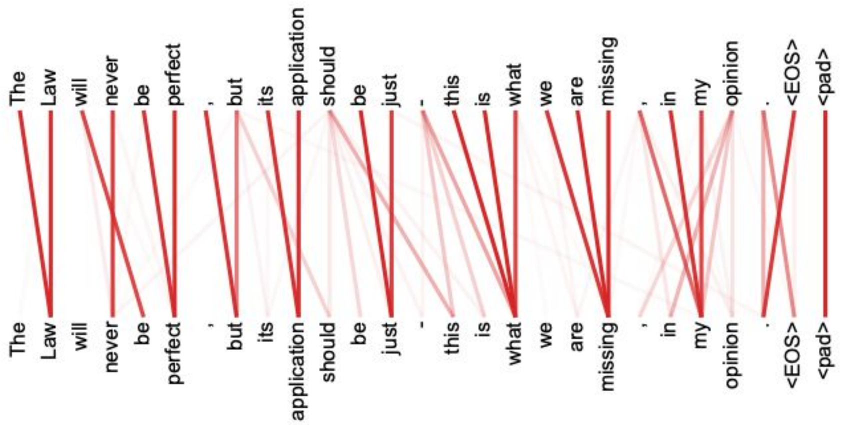
This is horribly complicated but the intuition is good: separate what h_t is good for. Some of it is good for right now (y_t); some of it is good for later!

Classic deep learning: add more layers!





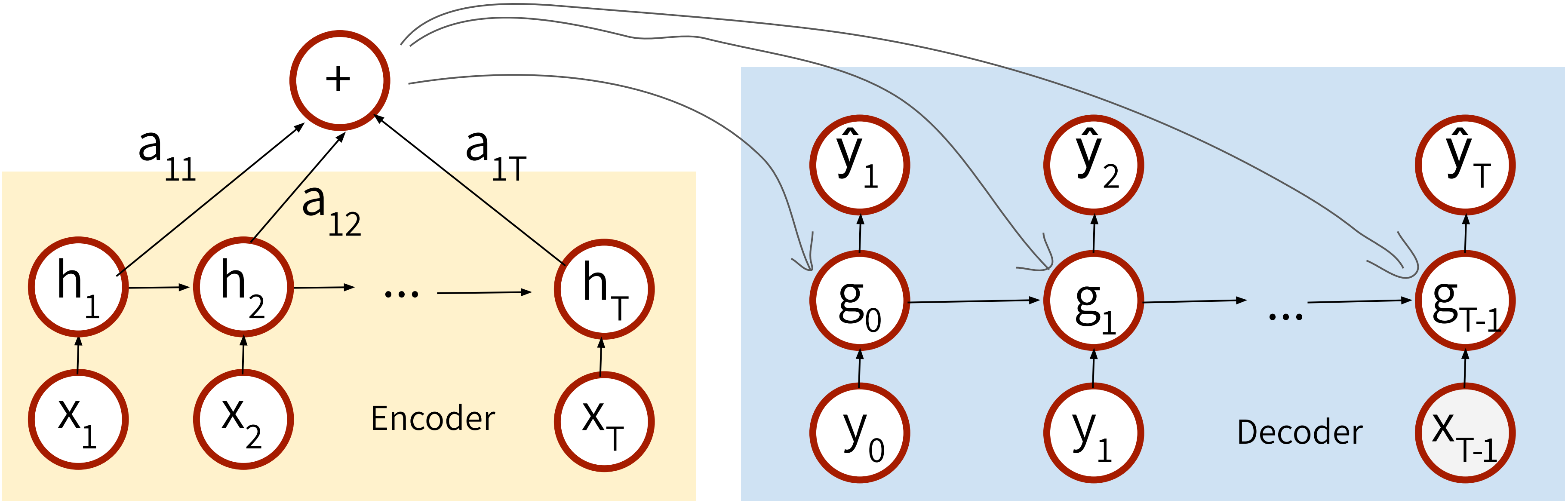
Regular Attention



Self Attention

Attention-Seq2seq

Intuition: Every timestep in decoder has its own attention to $h_1, \ldots, h_T$



- Suppose we are at timestep **i** in the decoder.
- Use g_{i-1} as query. Also $h_1, \ldots, h_T$ doubles as keys and values (self-attn)!

$$a_{it} = \frac{\exp\{\, sim(g_{i-1}, h_t) \,\}}{\sum_{s=1}^{T} \exp\{\, sim(g_{i-1}, h_s) \,\}}$$

$$c_i = \sum_{t=1}^{T} a_{it}\, h_t$$

$$g_i = decoder\_rnn(y_i, g_{i-1}, c_i)$$