CS 224S / LINGUIST 285
Spoken Language Processing

Mike Wu, Andrew Maas
Stanford University

Lecture 6: Deep Learning Preliminaries
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’.

Outputs $p(y=1|x_1...x_n) = \sigma(z)$, the probability of predicting class 1!
Multi-layer Perceptrons

Stack logistic units!

Hidden layer
Multi-layer Perceptrons

Forward Propagation

\[ z_{j}^{(l+1)} = \sum w_{ij}^{(l)} a_{i}^{(l)} + b_{j}^{(l)} \]

\[ h_{j}^{(l)} = \sigma(z_{j}^{(l)}) \quad \text{“activation”} \]

\[ \theta = \{ w_{ij}^{(l)}, b_{j}^{(l)} \text{ for all } i,j,l \} \]

\( \wedge \text{ “parameters”} \)
“Deep” Neural Networks

Forward pass works the same way
Objective Function

 Depends on the task! Examples:

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

Now we have to optimize!
Let’s do $\frac{dJ}{dw_{11}^{(0)}}$ 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.
Encoder-Decoder Models
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 \(y_t\)

\[
\hat{y}_t = \{U, V, W\}
\]

\[
\begin{align*}
\hat{y}_{t-1} &\rightarrow h_{t-1} \\
h_t &\rightarrow h_t \\
x_t &\rightarrow h_t \\
\hat{y}_t &\rightarrow \hat{y}_t \\
\hat{y}_{t+1} &\rightarrow \hat{y}_{t+1}
\end{align*}
\]

\[
\begin{align*}
\hat{y}_{t-1} &\rightarrow \hat{y}_{t-1} \\
h_{t-1} &\rightarrow h_{t-1} \\
x_{t-1} &\rightarrow h_{t-1} \\
\hat{y}_t &\rightarrow \hat{y}_t \\
\hat{y}_{t+1} &\rightarrow \hat{y}_{t+1}
\end{align*}
\]

\[
\begin{align*}
\hat{y}_{t-1} &\rightarrow \hat{y}_{t-1} \\
h_{t-1} &\rightarrow h_{t-1} \\
x_{t-1} &\rightarrow h_{t-1} \\
\hat{y}_t &\rightarrow \hat{y}_t \\
\hat{y}_{t+1} &\rightarrow \hat{y}_{t+1}
\end{align*}
\]

\[
\begin{align*}
\hat{y}_{t-1} &\rightarrow \hat{y}_{t-1} \\
h_{t-1} &\rightarrow h_{t-1} \\
x_{t-1} &\rightarrow h_{t-1} \\
\hat{y}_t &\rightarrow \hat{y}_t \\
\hat{y}_{t+1} &\rightarrow \hat{y}_{t+1}
\end{align*}
\]

\[
\begin{align*}
\hat{y}_{t-1} &\rightarrow \hat{y}_{t-1} \\
h_{t-1} &\rightarrow h_{t-1} \\
x_{t-1} &\rightarrow h_{t-1} \\
\hat{y}_t &\rightarrow \hat{y}_t \\
\hat{y}_{t+1} &\rightarrow \hat{y}_{t+1}
\end{align*}
\]

\[
\begin{align*}
\hat{y}_{t-1} &\rightarrow \hat{y}_{t-1} \\
h_{t-1} &\rightarrow h_{t-1} \\
x_{t-1} &\rightarrow h_{t-1} \\
\hat{y}_t &\rightarrow \hat{y}_t \\
\hat{y}_{t+1} &\rightarrow \hat{y}_{t+1}
\end{align*}
\]

\[
\begin{align*}
\hat{y}_{t-1} &\rightarrow \hat{y}_{t-1} \\
h_{t-1} &\rightarrow h_{t-1} \\
x_{t-1} &\rightarrow h_{t-1} \\
\hat{y}_t &\rightarrow \hat{y}_t \\
\hat{y}_{t+1} &\rightarrow \hat{y}_{t+1}
\end{align*}
\]

\[
\begin{align*}
\hat{y}_{t-1} &\rightarrow \hat{y}_{t-1} \\
h_{t-1} &\rightarrow h_{t-1} \\
x_{t-1} &\rightarrow h_{t-1} \\
\hat{y}_t &\rightarrow \hat{y}_t \\
\hat{y}_{t+1} &\rightarrow \hat{y}_{t+1}
\end{align*}
\]
U, V, W are shared over all timesteps (same ones!)

The model does a forward pass for every single timestep: $f_\theta(x_t, h_{t-1})$

Objective (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]
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.
Input is a sequence of tokens \((x_1, x_2, \ldots, x_T)\), no output sequence. The model should learn \(p(x_t | x_1, \ldots, x_{t-1})\).

**RNN Decoder**

Training:

```
\text{Input} \xrightarrow{W} \text{h}_{t-1} \xrightarrow{V} \text{X}_t \xrightarrow{U} \text{h}_t
```

Generation:

```
\text{h}_{t-1} \xrightarrow{W} \text{h}_t \xrightarrow{V} \text{X}_t
```

"Teacher forcing"
All that work and we have our first encoder-decoder model!

Given an input and output sequence \((x_1, x_2, \ldots, x_t), (y_1, y_2, \ldots, y_{t'})\), the model should capture \(p(y_t | y_1, \ldots, y_{t-1}, x_1, \ldots, x_T)\). It gets to see all of \(x\)!

**Seq2Seq** [Cho et al. 2014, Susekever et al. 2014]
Two different RNNs glue’d together (separate parameters)

- One of them encodes \((x_1, \ldots, x_T)\) 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 \((x = \text{speaker 1}, y = \text{speaker 2})\)
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).

**Speaker Consistency**

- Where do you live now?
  - I live in Los Angeles.

- In which city do you live now?
  - I live in Paris.

- In which country do you live now?
  - England, you?
Vanishing gradient problem & local dependency of RNNs.
- Attention gets at this more directly (and simply).

**Holy grail:** capturing long term dependencies.

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

**Intuition:**

- Decoder
- Weights sum to 1!
Notation: (general, we will revisit seq2seq)

\[ q \in \mathbb{R}^d: \text{query} ; \ k_1, \ldots, k_T \in \mathbb{R}^d: \text{keys} ; \ v_1, \ldots, v_T \in \mathbb{R}^d: \text{values} \]
**Notation:** (general, we will revisit seq2seq)

\[ q \in \mathbb{R}^d: \text{query} \; ; \; k_1, \ldots, k_T \in \mathbb{R}^d: \text{keys} \; ; \; v_1, \ldots, v_T \in \mathbb{R}^d: \text{values} \]

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

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

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

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

\[
\text{sim}(q, k_t) = q^T k_t / \sqrt{d} \quad \text{[Luong et. al. 2015]} \quad \text{(scaled dot-pdt)}
\]
**Notation:** (general, we will revisit seq2seq)

\( q \in \mathbb{R}^d: \text{query} \; ; \; k_1, \ldots, k_T \in \mathbb{R}^d: \text{keys} \; ; \; v_1, \ldots, v_T \in \mathbb{R}^d: \text{values} \)

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

**Step 2:** compute attention weights \( a_t. \)

\[
a_t = \frac{\exp\{ \text{sim}(q, k_t) \}}{\sum_{s=1}^{T} \exp\{ \text{sim}(q, k_s) \}}
\]

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

Also \( \sum_t a_t = 1 \)
Notation: (general, we will revisit seq2seq)

\( q \in \mathbb{R}^d: \text{query} \); \( k_1, \ldots, k_T \in \mathbb{R}^d: \text{keys} \); \( v_1, \ldots, v_T \in \mathbb{R}^d: \text{values} \)

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

**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!?

<table>
<thead>
<tr>
<th>Method</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-based</td>
<td>This is my big red <strong>dog</strong>, Clifford.</td>
</tr>
<tr>
<td>Description-based</td>
<td>This is my big <strong>red</strong> dog, Clifford.</td>
</tr>
<tr>
<td>Reference-based</td>
<td>This is my <strong>big red</strong> dog, Clifford.</td>
</tr>
</tbody>
</table>

What’s useful depends on the task. How do I pick what to do?
Idea [Vaswani et. al. 2017]: Don’t pick. Pay attention as if you had “multiple heads”.

Multi-Head Attention

Linear

Concat

Scaled Dot-Product Attention

Linear

Linear

Linear

V

K

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

Pick $H$ heads.

For $h$ in range($H$):

$$q^{(h)} = \text{MLP}_h (q)$$
$$k^{(h)}_t = \text{MLP}_h (k_t) \text{ for all } t$$
$$v^{(h)}_t = \text{MLP}_h (v_t) \text{ for all } t$$
Idea [Vaswani et. al. 2017]: Don’t pick. Pay attention as if you had “multiple heads”.

Pick H heads.

For h in range(H):

\[ q^{(h)} = MLP_h(q) \]
\[ k^{(h)} = MLP_h(k_t) \text{ for all } t \]
\[ v^{(h)}_t = MLP_h(v_t) \text{ for all } t \]

\[ a^{(h)}_t = \frac{\exp\{ q^{(h)^T}k^{(h)}_t / \sqrt{d} \} }{\sum_{s=1}^{T} \exp\{ q^{(h)^T}k^{(h)}_s / \sqrt{d} \} } \]

\[ c^{(h)} = \sum_{t=1}^{T} a^{(h)}_t v^{(h)}_t \]
Idea [Vaswani et. al. 2017]: Don’t pick. Pay attention as if you had “multiple heads”.

Pick $H$ heads.

For $h$ in range($H$):

$$
\begin{align*}
q^{(h)} &= \text{MLP}_h(q) \\
k^{(h)} &= \text{MLP}_h(k_t) \text{ for all } t \\
v^{(h)}_t &= \text{MLP}_h(v_t) \text{ for all } t \\
\alpha^{(h)}_t &= \frac{\exp\{ q^{(h)T}k^{(h)}_t / \sqrt{d} \}}{\sum_{s=1}^{T} \exp\{ q^{(h)T}k^{(h)}_s / \sqrt{d} \}} \\
c^{(h)} &= \sum_{t=1}^{T} \alpha^{(h)}_t v^{(h)}_t \\
c_{all} &= \text{concat}(c^{(1)}, \ldots, c^{(H)}) \\
c &= \text{linear}(c_{all}) \quad \# \text{ project to smaller dimension}
\end{align*}
$$
Self Attention [Vaswani et. al. 2017]

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

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

\[ q_t = \text{MLP}(x_t), \quad k_t = \text{MLP}(x_t), \quad v_t = \text{MLP}(x_t) \]

Then we can extract a sequence:

\[ c_t = \sum_{t=1}^{T} a_t v_t \quad \Rightarrow \quad (c_1, c_2, ..., c_T) \]
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)!

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

\[
c_i = \sum_{t=1}^{T} a_{it} h_t
\]

\[
g_i = \text{decoder_rnn}(y_i, g_{i-1}, c_i)
\]
Transformers (SOTA)
A transformer layer is composed of an encoder and a decoder. 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

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

$(e_1, e_2, \ldots, e_T)$ e.g. mel spectrogram
We’ve seen this! We do self-attention with $H$ heads on the lookup embeddings.

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

Outputs $(c_1, c_2, \ldots, c_T)$ each $c_t \in \mathbb{R}^d$
Residual sum

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

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

$h_t^{\text{res}} = c_t + e_t$ residual output
Layer normalization

\[ h_{\text{norm}} = \frac{h^{\text{res}} - E[h^{\text{res}}]}{\sqrt{\text{Var}[h^{\text{res}}] + \epsilon}} \times \gamma + \beta \]

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

\[ h^{\text{res}} = (h_1^{\text{res}}, h_2^{\text{res}}, \ldots, h_T^{\text{res}}). \]

- 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.
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:

\[
\begin{align*}
PE(t, 2i) & = \sin(t/10000^{2i/d}) \\
PE(t, 2i+1) & = \cos(t/10000^{2i/d}) \\
PE(t) & = [PE(t, 0), PE(t, 1), \ldots, PE(t, d)]
\end{align*}
\]

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

\[
\begin{align*}
PE(t, 2i) &= \sin\left(t/10000^{2i/d}\right) \\
PE(t, 2i+1) &= \cos\left(t/10000^{2i/d}\right)
\end{align*}
\]

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

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 positional encoding for a sentence with the maximum length of 50. Each row represents the embedding vector $\vec{p}_t$. 

https://kazemnejad.com/blog/transformer_architecture_positional_encoding
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.

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

Cheating if we see this b/c in test time, we don’t have access to > t+1
Masked Multi-head Attention:

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

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

Transformer decoder
Encoder Multi-head Attention:

- Output of encoder: \((h^{\text{enc}}_1, h^{\text{enc}}_2, \ldots, h^{\text{enc}}_T)\).
- 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!
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
Starting thinking about project ideas!
Resources


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
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
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

**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.

**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

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]

A Solution: Auxiliary objectives!

Optimize for high mutual information between source and response!

\[ J = -\log p(\text{“I don’t know”} \mid \text{“how’s life”}) + \text{MI(“i don’t know, “how’s life”)} \]

Regular objective \hspace{1cm} \text{Regularization} \sim 0
Problem: Just English sounding responses isn’t enough. Chatbots should not contradict themselves factually.
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.
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.

Create new independent variables in forward pass:

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

\[ \dot{w}_1 = \frac{dw_1}{dx_1} \]

want \( \frac{df}{dx_1} \) so \( w_1 = 1 \)

product rule

\[ \dot{w}_1 = 1 \text{ (seed)} \]

\[ \dot{w}_2 = 0 \text{ (seed)} \]

chain rule + sine rule

\[ \dot{w}_3 = w_2 \cdot \dot{w}_1 + w_1 \cdot \dot{w}_2 \]

\[ \dot{w}_4 = \cos w_1 \cdot \dot{w}_1 \]

\[ \dot{w}_5 = \dot{w}_3 + \dot{w}_4 \]

https://en.wikipedia.org/wiki/Automatic_differentiation
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})] Loss: {:.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
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 \( T \to \infty \). Say we want to calculate \( \frac{dy_t}{dx_1} \).

What happens if \( \frac{dh_t}{dh_{t-1}} < 1 \) for all \( t \)?

**Impact of \( x_{t-s} \) on \( y_t \) decreases as \( s \) increases.**
**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”

\[
    k_t = c_{t-1} \cdot f_t
\]

- Do the “forgetting”

**Add gate:**

\[
    g_t = \sigma(U_g h_{t-1} + W_g x_t)
\]

- Usual RNN function

\[
    l_t = \sigma(U_i h_{t-1} + W_i x_t)
\]

- Pick what to “add”

\[
    j_t = g_t \cdot i_t
\]

- Do the “adding”

\[
    c_t = j_t + k_t
\]

- context is some of last context and some new stuff
Forget gate: \[ f_t = \sigma(U_f h_{t-1} + W_f x_t) \quad \text{Pick what to “forget”} \]
\[ k_t = c_{t-1} * f_t \quad \text{Do the “forgetting”} \]

Add gate: \[ g_t = \sigma(U_g h_{t-1} + W_g x_t) \quad \text{Usual RNN function} \]
\[ l_t = \sigma(U_i h_{t-1} + W_i x_t) \quad \text{Pick what to “add”} \]
\[ j_t = g_t * i_t \quad \text{Do the “adding”} \]
\[ c_t = j_t + k_t \quad \text{context is some of last context and some new stuff} \]

Output gate: \[ o_t = \sigma(U_o h_{t-1} + W_o x_t) \quad \text{Pick what to use for current timestep} \]
\[ H_t = o_t * \tanh(c_t) \quad \text{Do the partitioning} \]
Forget gate:
\[ f_t = \sigma(U_f h_{t-1} + W_f x_t) \]  
Pick what to “forget”

\[ k_t = c_{t-1} \times f_t \]  
Do the “forgetting”

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

\[ l_t = \sigma(U_i h_{t-1} + W_i x_t) \]  
Pick what to “add”

\[ j_t = g_t \times i_t \]  
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

\[ H_t = o_t \times \tanh(c_t) \]  
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!
Improvements to RNN

Stacked RNNs

Bi-directional RNNs
Classic deep learning: add more layers!

“sequence classification mode”

“Language model mode”
Regular Attention

Self Attention