Next several slides are from Carlos Guestrin, Luke Zettlemoyer
Human Neurons

- **Switching time**
  - ~ 0.001 second
- **Number of neurons**
  - $10^{10}$
- **Connections per neuron**
  - $10^{4.5}$
- **Scene recognition time**
  - 0.1 seconds
- **Number of cycles per scene recognition?**
  - 100 → much parallel computation!
Perceptron as a Neural Network

This is one neuron:

- Input edges $x_1 \ldots x_n$, along with basis $x_0 = 1$
- The sum is represented graphically
- Sum passed through an activation function $g$

$$g = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
-1 & \text{otherwise}
\end{cases}$$
Sigmoid Neuron

\[
g(w_0 + \sum_i w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}
\]

Just change \( g \)!
- Why would we want to do this?
- Notice new output range \([0,1]\). What was it before?
- Look familiar?
Optimizing a neuron

We train to minimize sum-squared error

\[
\ell(W) = \frac{1}{2} \sum_j [y^j - g(w_0 + \sum_i w_ix_i^j)]^2
\]

\[
\frac{\partial \ell}{\partial w_i} = - \sum_j [y_j - g(w_0 + \sum_i w_ix_i^j)] \frac{\partial}{\partial w_i} g(w_0 + \sum_i w_ix_i^j)
\]

\[
\frac{\partial}{\partial w_i} g(w_0 + \sum_i w_ix_i^j) = x_i^j g'(w_0 + \sum_i w_ix_i^j)
\]

\[
\frac{\partial \ell(W)}{\partial w_i} = - \sum_j [y^j - g(w_0 + \sum_i w_ix_i^j)] x_i^j g'(w_0 + \sum_i w_ix_i^j)
\]

Solution just depends on \(g':\) derivative of activation function!
Sigmoid units: have to differentiate

$$\frac{\partial \ell(W)}{\partial w_i} = - \sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)] x_i^j g'(w_0 + \sum_i w_i x_i^j)$$

$$g(x) = \frac{1}{1 + e^{-x}} \quad g'(x) = g(x)(1 - g(x))$$

$$\delta^j = [y^j - g(w_0 + \sum_i w_i x_i^j)] g^j (1 - g^j)$$

$$g^j = g(w_0 + \sum_i w_i x_i^j)$$

$$w_i \leftarrow w_i + \eta \sum_j x_i^j \delta^j$$
Perceptron, linear classification, Boolean functions: $x_i \in \{0, 1\}$

- Can learn $x_1 \lor x_2$?
  - $-0.5 + x_1 + x_2$
- Can learn $x_1 \land x_2$?
  - $-1.5 + x_1 + x_2$
- Can learn any conjunction or disjunction?
  - $-0.5 + x_1 + \ldots + x_n$
  - $(-n + 0.5) + x_1 + \ldots + x_n$
- Can learn majority?
  - $(-0.5 \times n) + x_1 + \ldots + x_n$
- What are we missing? The dreaded XOR!, etc.
Going beyond linear classification

Solving the XOR problem

\[ y = x_1 \text{ XOR } x_2 = (x_1 \land \neg x_2) \lor (x_2 \land \neg x_1) \]

\[ v_1 = (x_1 \land \neg x_2) \]
\[ = -1.5 + 2x_1 - x_2 \]
\[ v_2 = (x_2 \land \neg x_1) \]
\[ = -1.5 + 2x_2 - x_1 \]
\[ y = v_1 \lor v_2 \]
\[ = -0.5 + v_1 + v_2 \]
Hidden layer

- **Single unit:**

\[ \text{out}(x) = g(w_0 + \sum_i w_i x_i) \]

- **1-hidden layer:**

\[ \text{out}(x) = g \left( w_0 + \sum_k w_k g(w_0^k + \sum_i w_i^k x_i) \right) \]

- **No longer convex function!**
Example data for NN with hidden layer

A target function:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>

Can this be learned??
A network:

Learned weights for hidden layer

Learned hidden layer representation:

<table>
<thead>
<tr>
<th>Input Values</th>
<th>Hidden Values</th>
<th>Output Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000000</td>
<td>.89 .04 .08</td>
<td>100000000</td>
</tr>
<tr>
<td>010000000</td>
<td>.01 .11 .88</td>
<td>010000000</td>
</tr>
<tr>
<td>001000000</td>
<td>.01 .97 .27</td>
<td>001000000</td>
</tr>
<tr>
<td>000100000</td>
<td>.99 .97 .71</td>
<td>000100000</td>
</tr>
<tr>
<td>000010000</td>
<td>.03 .05 .02</td>
<td>000010000</td>
</tr>
<tr>
<td>000001000</td>
<td>.22 .99 .99</td>
<td>000001000</td>
</tr>
<tr>
<td>000000100</td>
<td>.80 .01 .98</td>
<td>000000100</td>
</tr>
<tr>
<td>000000010</td>
<td>.60 .94 .01</td>
<td>000000010</td>
</tr>
<tr>
<td>000000001</td>
<td>.80 .94 .01</td>
<td>000000001</td>
</tr>
</tbody>
</table>
Why “representation learning”?

• MaxEnt (multinomial logistic regression):

\[ y = \text{softmax}(w \cdot f(x, y)) \]

You design the feature vector

• NNs:

\[ y = \text{softmax}(w \cdot \sigma(Ux)) \]

\[ y = \text{softmax}(w \cdot \sigma(U^{(n)}(...\sigma(U^{(2)}\sigma(U^{(1)}x)))))) \]

Feature representations are “learned” through hidden layers
Very deep models in computer vision

\(^1\)Inception 5 (GoogLeNet)

Inception 7a

\(^1\)Going Deeper with Convolutions, [C. Szegedy et al, CVPR 2015]
RECURRENT NEURAL NETWORKS
Recurrent Neural Networks (RNNs)

- Each RNN unit computes a new hidden state using the previous state and a new input $$h_t = f(x_t, h_{t-1})$$
- Each RNN unit (optionally) makes an output using the current hidden state $$y_t = \text{softmax}(Vh_t)$$

- Hidden states $$h_t \in \mathbb{R}^D$$ are continuous vectors
  - Can represent very rich information
  - Possibly the entire history from the beginning
- Parameters are shared (tied) across all RNN units (unlike feedforward NNs)
Recurrent Neural Networks (RNNs)

- **Generic RNNs:**
  \[ h_t = f(x_t, h_{t-1}) \]
  \[ y_t = \text{softmax}(Vh_t) \]

- **Vanilla RNN:**
  \[ h_t = \tanh(Ux_t + Wh_{t-1} + b) \]
  \[ y_t = \text{softmax}(Vh_t) \]
Tanh

- Often used for hidden states & cells in RNNs, LSTMs
- Pro: differentiable, often converges faster than sigmoid
- Con: gradients easily saturate to zero => vanishing gradients

\[ \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \]

\[ \tanh'(x) = 1 - \tanh^2(x) \]

\[ \tanh(x) = 2\sigma(2x) - 1 \]
Sigmoid

- Often used for gates
- Pro: neuron-like, differentiable
- Con: gradients saturate to zero almost everywhere except $x$ near zero $\Rightarrow$ vanishing gradients
- Batch normalization helps

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$
Recurrent Neural Networks (RNNs)

- Generic RNNs: \( h_t = f(x_t, h_{t-1}) \)
- Vanilla RNNs: \( h_t = \tanh(U x_t + W h_{t-1} + b) \)
- LSTMs (Long Short-term Memory Networks):\
  \[\
  i_t = \sigma(U^{(i)} x_t + W^{(i)} h_{t-1} + b^{(i)}) \\
  f_t = \sigma(U^{(f)} x_t + W^{(f)} h_{t-1} + b^{(f)}) \\
  o_t = \sigma(U^{(o)} x_t + W^{(o)} h_{t-1} + b^{(o)}) \\
  \tilde{c}_t = \tanh(U^{(c)} x_t + W^{(c)} h_{t-1} + b^{(c)}) \\
  c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
  h_t = o_t \circ \tanh(c_t) \\
  \]

There are many known variations to this set of equations!
Many uses of RNNs
1. Classification (seq to one)

- Input: a sequence
- Output: one label (classification)
- Example: sentiment classification

\[ h_t = f(x_t, h_{t-1}) \]
\[ y = \text{softmax}(Vh_n) \]
Many uses of RNNs

2. One to seq

- Input: one item
- Output: a sequence
- Example: Image captioning

\[ h_t = f(x_t, h_{t-1}) \]
\[ y_t = \text{softmax}(V h_t) \]
Many uses of RNNs
3. sequence tagging

- Input: a sequence
- Output: a sequence (of the same length)
- Example: POS tagging, Named Entity Recognition
- How about Language Models?
  - Yes! RNNs can be used as LMs!
  - RNNs make Markov assumption: T/F?

\[
\begin{align*}
h_t &= f(x_t, h_{t-1}) \\
y_t &= \text{softmax}(Vh_t)
\end{align*}
\]
Many uses of RNNs
4. Language models

- Input: a sequence of words
- Output: one next word
- Output: or a sequence of next words
- During training or if used for measuring LM score, $x_t$ is the actual word in the training sentence.
- If used for sampling, $x_t$ is the word predicted from the previous time step.
- Does RNN LMs make Markov assumption? 
  - i.e., the next word depends only on the previous $N$ words?

$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(Vh_t)$$
Many uses of RNNs

5. **seq2seq** (aka “encoder-decoder”)

- Input: a sequence
- Output: a sequence (of *different* length)
- Examples?

\[
\begin{align*}
    h_t &= f(x_t, h_{t-1}) \\
    y_t &= \text{softmax}(V h_t)
\end{align*}
\]
Many uses of RNNs

4. seq2seq (aka “encoder-decoder”)

- Conversation and Dialogue
- Machine Translation

Figure from http://www.wildml.com/category/conversational-agents/
Many uses of RNNs

4. seq2seq (aka “encoder-decoder”)

Parsing!
- “Grammar as Foreign Language” (Vinyals et al., 2015)

(S (NP NNP)_{NP} (VP VBZ (NP DT NN)_{NP})_{VP} \cdot )_{S}

John has a dog
Hafez: Neural Sonnet Writer
(Ghazvininejad et al. 2016)
Neural Sonnets

Deep Convolution Network
Outrageous channels on the wrong connections,
An empty space without an open layer,
A closet full of black and blue extensions,
Connections by the closure operator.

Theory
Another way to reach the wrong conclusion!
A vision from a total transformation,
Created by the great magnetic fusion,
Lots of people need an explanation.
Recurrent Neural Networks (RNNs)

- Generic RNNs: \( h_t = f(x_t, h_{t-1}) \)
  \( y_t = \text{softmax}(Vh_t) \)

- Vanilla RNN: \( h_t = \tanh(Ux_t + Wh_{t-1} + b) \)
  \( y_t = \text{softmax}(Vh_t) \)
Recurrent Neural Networks (RNNs)

- **Generic RNNs:**  
  \[ h_t = f(x_t, h_{t-1}) \]

- **Vanilla RNNs:**  
  \[ h_t = \tanh(Ux_t + Wh_{t-1} + b) \]

- **LSTMs (Long Short-term Memory Networks):**  
  \[
  \begin{align*}
  i_t &= \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)}) \\
  f_t &= \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)}) \\
  o_t &= \sigma(U^{(o)}x_t + W^{(o)}h_{t-1} + b^{(o)}) \\
  \tilde{c}_t &= \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)}) \\
  c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
  h_t &= o_t \circ \tanh(c_t)
  \end{align*}
  \]

There are many known variations to this set of equations!
LSTMs (LONG SHORT-TERM MEMORY NETWORKS)

Figure by Christopher Olah (colah.github.io)
LSTMs (LONG SHORT-TERM MEMORY NETWORKS)

sigmoid: [0,1]

Forget gate: forget the past or not

\[ f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)}) \]
**LSTMs (LONG SHORT-TERM MEMORY NETWORKS)**

sigmoid: $[0,1]$

忘却门：忘记过去的或不

$$f_t = \sigma(U(f)x_t + W(f)h_{t-1} + b(f))$$

输入门：使用输入的或不

$$i_t = \sigma(U(i)x_t + W(i)h_{t-1} + b(i))$$

新细胞内容（temp）：

$$\tilde{c}_t = \text{tanh}(U(c)x_t + W(c)h_{t-1} + b(c))$$
LSTMs (LONG SHORT-TERM MEMORY NETWORKS)

sigmoid: [0,1]

\[ f_t = \sigma(U^{(f)} x_t + W^{(f)} h_{t-1} + b^{(f)}) \]

\[ i_t = \sigma(U^{(i)} x_t + W^{(i)} h_{t-1} + b^{(i)}) \]

\[ \tilde{c}_t = \tanh(U^{(c)} x_t + W^{(c)} h_{t-1} + b^{(c)}) \]

\[ c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \]

Figure by Christopher Olah (colah.github.io)
LSTMs (LONG SHORT-TERM MEMORY NETWORKS)

Output gate: output from the new cell or not
\[ o_t = \sigma(U^{(o)}x_t + W^{(o)}h_{t-1} + b^{(o)}) \]

Hidden state:
\[ h_t = o_t \circ \tanh(c_t) \]

Forget gate: forget the past or not
\[ f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)}) \]

Input gate: use the input or not
\[ i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)}) \]

New cell content (temp):
\[ \tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)}) \]

New cell content:
- mix old cell with the new temp cell
\[ c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \]

Figure by Christopher Olah (colah.github.io)
LSTMs (LONG SHORT-TERM MEMORY NETWORKS)

Forget gate: forget the past or not
Input gate: use the input or not
Output gate: output from the new cell or not

New cell content (temp):
New cell content:
- mix old cell with the new temp cell

Hidden state:
\[ h_t = o_t \circ \tanh(c_t) \]

\[
\begin{align*}
    f_t &= \sigma(U^{(f)} x_t + W^{(f)} h_{t-1} + b^{(f)}) \\
    i_t &= \sigma(U^{(i)} x_t + W^{(i)} h_{t-1} + b^{(i)}) \\
    o_t &= \sigma(U^{(o)} x_t + W^{(o)} h_{t-1} + b^{(o)}) \\
    \tilde{c}_t &= \tanh(U^{(c)} x_t + W^{(c)} h_{t-1} + b^{(c)}) \\
    c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t
\end{align*}
\]
vanishing gradient problem for RNNs.

- The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity).
- The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network ‘forgets’ the first inputs.

Example from Graves 2012
Preservation of gradient information by LSTM

For simplicity, all gates are either entirely open (‘O’) or closed (‘—’).

The memory cell ‘remembers’ the first input as long as the forget gate is open and the input gate is closed.

The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.

Example from Graves 2012
Recurrent Neural Networks (RNNs)

- **Generic RNNs:** \( h_t = f(x_t, h_{t-1}) \)
- **Vanilla RNNs:** \( h_t = \tanh(Ux_t + Wh_{t-1} + b) \)
- **GRUs (Gated Recurrent Units):** \(^*\) (Cho et al, 2014)

\[
\begin{align*}
z_t &= \sigma(U^{(z)}x_t + W^{(z)}h_{t-1} + b^{(z)}) \\
r_t &= \sigma(U^{(r)}x_t + W^{(r)}h_{t-1} + b^{(r)}) \\
\tilde{h}_t &= \tanh(U^{(h)}x_t + W^{(h)}(r_t \circ h_{t-1}) + b^{(h)}) \\
h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t
\end{align*}
\]

---

*Less parameters than LSTMs. Easier to train for comparable performance!*
RNN Learning: **Backprop Through Time** (BPTT)

- Similar to backprop with non-recurrent NNs
- But unlike feedforward (non-recurrent) NNs, each unit in the computation graph repeats the exact same parameters…
- Backprop gradients of the parameters of each unit as if they are different parameters
- When updating the parameters using the gradients, use the average gradients throughout the entire chain of units.
Gates

• Gates contextually control information flow
• Open/close with sigmoid
• In LSTMs and GRUs, they are used to (contextually) maintain longer term history
**Bi-directional RNNs**

- Can incorporate context from both directions
- Generally improves over uni-directional RNNs
Google NMT (Oct 2016)
Tree LSTMs

- Are tree LSTMs more expressive than sequence LSTMs?
- I.e., recursive vs recurrent

- When Are Tree Structures Necessary for Deep Learning of Representations?
  Jiwei Li, Minh-Thang Luong, Dan Jurafsky and Eduard Hovy. EMNLP, 2015.
Recursive Neural Networks

- Sometimes, inference over a tree structure makes more sense than sequential structure.
- An example of compositionality in ideological bias detection (red $\rightarrow$ conservative, blue $\rightarrow$ liberal, gray $\rightarrow$ neutral) in which modifier phrases and punctuation cause polarity switches at higher levels of the parse tree.

Example from Iyyer et al., 2014
Recursive Neural Networks

- NNs connected as a tree
- Tree structure is fixed a priori
- Parameters are shared, similarly as RNNs

Example from Iyyer et al., 2014
Neural Probabilistic Language Model  (Bengio 2003)

\[ i\text{-th output} = P(w_t = i \mid \text{context}) \]

most computation here

\[ \text{tanh} \]

\[ C(w_{t-n+1}) \]

Table look-up in C

index for \( w_{t-n+1} \)

\[ C(w_{t-2}) \]

Matrix C

shared parameters across words

index for \( w_{t-2} \)

\[ C(w_{t-1}) \]

index for \( w_{t-1} \)
Neural Probabilistic Language Model  (Bengio 2003)

- Each word prediction is a separate feed forward neural network
- Feedforward NNLM is a Markovian language model
- Dashed lines show optional direct connections

\[
NN_{DMLP1}(x) = [\tanh(xW^1 + b^1), x]W^2 + b^2
\]

- \( W^1 \in \mathbb{R}^{d_{in} \times d_{hid}} \), \( b^1 \in \mathbb{R}^{1 \times d_{hid}} \); first affine transformation
- \( W^2 \in \mathbb{R}^{(d_{hid}+d_{in}) \times d_{out}} \), \( b^2 \in \mathbb{R}^{1 \times d_{out}} \); second affine transformation
ATTENTION!
Encoder – Decoder Architecture

Sequence-to-Sequence

Diagram borrowed from Alex Rush
Trial: Hard Attention

- At each step generating the target word $s^t_i$
- Compute the best alignment to the source word $s^s_j$
- And incorporate the source word to generate the target word

$$y^t_i = \arg\max_y O(y, s^t_i, s^s_j)$$

- Contextual hard alignment. How?

$$z_j = \tanh([s^t_i, s^s_j]W + b)$$
$$j = \arg\max_z z_j$$

- Problem?
Attention: Soft Alignments

- At each step generating the target word $s^t_i$
- Compute the attention $c$ to the source sequence $s^s$
- And incorporate the attention to generate the target word

$$y^t_i = \text{argmax}_y O(y, s^t_i, s^s_j)$$

- Contextual attention as soft alignment. How?

$$z_j = \tanh([s^t_i, s^s_j]W + b)$$
$$\alpha = \text{softmax}(z)$$
$$c = \sum_j \alpha_j s^s_j$$

- Step-1: compute the attention weights
- Step-2: compute the attention vector as interpolation
Attention

Diagram borrowed from Alex Rush
Attention parameterization

- Feedforward NNs
  \[ z_j = \tanh([s_i^t; s_j^s]W + b) \]
  \[ z_j = \tanh([s_i^t; s_j^s; s_i^t \odot s_j^s]W + b) \]

- Dot product
  \[ z_j = s_i^t \cdot s_j^s \]

- Cosine similarity
  \[ z_j = \frac{s_i^t \cdot s_j^s}{||s_i^t|| \cdot ||s_j^s||} \]

- Bi-linear models
  \[ z_j = s_i^T W s_j^s \]
Learned Attention!

Diagram borrowed from Alex Rush
Qualitative results

Figure 2. Attention over time. As the model generates each word, its attention changes to reflect the relevant parts of the image. “soft” (top row) vs “hard” (bottom row) attention. (Note that both models generated the same captions in this example.)

Figure 3. Examples of attending to the correct object (white indicates the attended regions, underlines indicated the corresponding word)
BiDAF
LEARNING: TRAINING DEEP NETWORKS
Vanishing / exploding Gradients

• Deep networks are hard to train
• Gradients go through multiple layers
• The multiplicative effect tends to lead to exploding or vanishing gradients
• Practical solutions w.r.t.
  – network architecture
  – numerical operations
Vanishing / exploding Gradients

- Practical solutions w.r.t. network architecture
  - Add skip connections to reduce distance
    - Residual networks, highway networks, ...
  - Add gates (and memory cells) to allow longer term memory
    - LSTMs, GRUs, memory networks, ...
Highway Network (Srivastava et al., 2015)

- A plain feedforward neural network:
  \[ y = H(x, W_H). \]
  - \( H \) is a typical affine transformation followed by a non-linear activation

- Highway network:
  \[ y = H(x, W_H) \cdot T(x, W_T) + x \cdot C(x, W_C). \]
  - \( T \) is a “transform gate”
  - \( C \) is a “carry gate”
  - Often \( C = 1 - T \) for simplicity
Residual Networks

• ResNet (He et al. 2015): first very deep (152 layers) network successfully trained for object recognition
Residual Networks

• Plain net

\[
\begin{align*}
x & \xrightarrow{\text{weight layer}} \xrightarrow{\text{relu}} H(x) \\
F(x) & \xrightarrow{\text{weight layer}} \xrightarrow{\text{relu}} H(x) = F(x) + x
\end{align*}
\]

• Residual net

\[
\begin{align*}
x & \xrightarrow{\text{weight layer}} \xrightarrow{\text{relu}} F(x) \\
H(x) & = F(x) + x
\end{align*}
\]

- F(x) is a residual mapping with respect to identity
- Direct input connection +x leads to a nice property w.r.t. back propagation --- more direct influence from the final loss to any deep layer
- In contrast, LSTMs & Highway networks allow for long distance input connection only through “gates”.
Residual Networks

Revolution of Depth

AlexNet, 8 layers (ILSVRC 2012)

VGG, 19 layers (ILSVRC 2014)

GoogleNet, 22 layers (ILSVRC 2014)

Residual Networks

Revolution of Depth

AlexNet, 8 layers (ILSVRC 2012)  
VGG, 19 layers (ILSVRC 2014)  
ResNet, 152 layers (ILSVRC 2015)

Residual Networks

Revolution of Depth

ImageNet Classification top-5 error (%)

Vanishing / exploding Gradients

• Practical solutions w.r.t. numerical operations
  – **Gradient Clipping**: bound gradients by a max value
  – **Gradient Normalization**: renormalize gradients when they are above a fixed norm
  – Careful initialization, smaller learning rates
  – Avoid saturating nonlinearities (like tanh, sigmoid)
    • ReLU or hard-tanh instead
Sigmoid

- Often used for gates
- Pro: neuron-like, differentiable
- Con: gradients saturate to zero almost everywhere except x near zero => vanishing gradients
- Batch normalization helps

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]

\[
\sigma'(x) = \sigma(x)(1 - \sigma(x))
\]
Tanh

- Often used for hidden states & cells in RNNs, LSTMs
- Pro: differentiable, often converges faster than sigmoid
- Con: gradients easily saturate to zero => vanishing gradients

\[
\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}
\]

\[
\tanh'(x) = 1 - \tanh^2(x)
\]

\[
\tanh(x) = 2\sigma(2x) - 1
\]
Hard Tanh

- **Pro:** computationally cheaper
- **Con:** saturates to zero easily, doesn’t differentiate at 1, -1

\[
\text{hardtanh}(t) = \begin{cases} 
-1 & t < -1 \\
 t & -1 \leq t \leq 1 \\
 1 & t > 1 
\end{cases}
\]
ReLU

- Pro: doesn’t saturate for \( x > 0 \), computationally cheaper, induces sparse NNs
- Con: non-differentiable at 0
- Used widely in deep NN, but not as much in RNNs
- We informally use subgradients:

\[
\text{ReLU}(x) = \max(0, x)
\]

\[
\frac{d \text{ReLU}(x)}{dx} = \begin{cases} 
1 & x > 0 \\
0 & x < 0 \\
1 \text{ or } 0 & \text{o.w}
\end{cases}
\]
Vanishing / exploding Gradients

• Practical solutions w.r.t. numerical operations
  – Gradient Clipping: bound gradients by a max value
  – Gradient Normalization: renormalize gradients when they are above a fixed norm
  – Careful initialization, smaller learning rates
  – Avoid saturating nonlinearities (like tanh, sigmoid)
    • ReLU or hard-tanh instead
  – Batch Normalization: add intermediate input normalization layers
Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...m\}$;  
Parameters to be learned: $\gamma$, $\beta$

**Output:** $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

\[
\begin{align*}
\mu_{\mathcal{B}} & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i & \quad & \text{// mini-batch mean} \\
\sigma_{\mathcal{B}}^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 & \quad & \text{// mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} & \quad & \text{// normalize} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) & \quad & \text{// scale and shift}
\end{align*}
\]
Regularization

• Regularization by objective term
  \[ \mathcal{L}(\theta) = \sum_{i=1}^{n} \max\{0, 1 - (\hat{y}_c - \hat{y}_c')\} + \lambda \|\theta\|^2 \]
  
  – Modify loss with L1 or L2 norms

• Less depth, smaller hidden states, early stopping

• Dropout
  – Randomly delete parts of network during training
  – Each node (and its corresponding incoming and outgoing edges) dropped with a probability p
  – P is higher for internal nodes, lower for input nodes
  – The full network is used for testing
  – Faster training, better results
  – Vs. Bagging
Convergence of backprop

- Without non-linearity or hidden layers, learning is convex optimization
  - Gradient descent reaches **global minima**
- **Multilayer neural nets (with nonlinearity)** are **not convex**
  - Gradient descent gets stuck in local minima
  - Selecting number of hidden units and layers = fuzzy process
  - NNs have made a HUGE comeback in the last few years
    - Neural nets are back with a new name
      - Deep belief networks
      - Huge error reduction when trained with lots of data on GPUs
SUPPLEMENTARY TOPICS
POINTER NETWORKS
Pointer Networks! (Vinyals et al. 2015)

- NNs with attention: content-based attention to input
- Pointer networks: location-based attention to input
- Applications: Convex haul, Delaunay Triangulation, Traveling Salesman

(a) Input $\mathcal{P} = \{P_1, \ldots, P_{10}\}$, and the output sequence $C^\mathcal{P} = \{\Rightarrow, 2, 4, 3, 5, 6, 7, 2, \Leftarrow\}$ representing its convex hull.

(b) Input $\mathcal{P} = \{P_1, \ldots, P_5\}$, and the output $C^\mathcal{P} = \{\Rightarrow, (1, 2, 4), (1, 4, 5), (1, 3, 5), (1, 2, 3), \Leftarrow\}$ representing its Delaunay Triangulation.
Figure 1:
(a) Sequence-to-Sequence - An RNN (blue) processes the input sequence to create a code vector that is used to generate the output sequence (purple) using the probability chain rule and another RNN. The output dimensionality is fixed by the dimensionality of the problem and it is the same during training and inference [1].

(b) Ptr-Net - An encoding RNN converts the input sequence to a code (blue) that is fed to the generating network (purple). At each step, the generating network produces a vector that modulates a content-based attention mechanism over inputs ([5, 2]). The output of the attention mechanism is a softmax distribution with dictionary size equal to the length of the input.

The main contributions of our work are as follows:

• We propose a new architecture, that we call Pointer Net, which is simple and effective. It deals with the fundamental problem of representing variable length dictionaries by using a softmax probability distribution as a "pointer".

• We apply the Pointer Net model to three distinct non-trivial algorithmic problems involving geometry. We show that the learned model generalizes to test problems with more points than the training problems.

• Our Pointer Net model learns a competitive small scale (n ≤ 50) TSP approximate solver. Our results demonstrate that a purely data driven approach can learn approximate solutions to problems that are computationally intractable.

2 Models
We review the sequence-to-sequence [1] and input-attention models [5] that are the baselines for this work in Sections 2.1 and 2.2. We then describe our model - Ptr-Net in Section 2.3.
Attention Mechanism vs Pointer Networks

\[
e_{ij} = v_a^T \tanh(W_a s_{i-1} + U_a h_j)
\]

\[
\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}
\]

\[
c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j
\]

Attention mechanism

\[
e_{ij} = v_a^T \tanh(W_a s_{i-1} + U_a h_j)
\]

\[
p(C_i|c_{1},...,c_{i-1},p) = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}
\]

Ptr-Net

Softmax normalizes the vector \(e_{ij}\) to be an output distribution over the dictionary of inputs.

Diagram borrowed from Keon Kim
CopyNet (Gu et al. 2016)

• Conversation
  – I: Hello Jack, my name is Chandralekha
  – R: Nice to meet you, Chandralekha
  – I: This new guy doesn’t perform exactly as expected.
  – R: what do you mean by “doesn’t perform exactly as expected?”

• Translation
CopyNet (Gu et al. 2016)

(a) Attention-based Encoder-Decoder (RNNSearch)

(b) Generate-Mode & Copy-Mode

\[ \text{Prob}(“\text{Jebara”}) = \text{Prob}(“\text{Jebara”}, g) + \text{Prob}(“\text{Jebara”}, c) \]

(c) State Update

\[ \rho \]

Figure 1: The overall diagram of CopyNet

CopyNet is a neural network model introduced by Gu et al. in 2016. It consists of three main components: (a) Attention-based Encoder-Decoder (RNNSearch), (b) Generate-Mode & Copy-Mode, and (c) State Update. The model aims to predict target sequences by copying source words or generating new ones. The diagram illustrates the flow of information through these components, highlighting how words from the source sequence can be copied or generated to form the target sequence. The model is depicted with arrows indicating the direction of information flow, with specific connections showing how the attention mechanism is used to focus on relevant parts of the input sequence.
CopyNet (Gu et al. 2016)

- Key idea: interpolation between generation model & copy model

\[ p(y_t|s_t, y_{t-1}, c_t, M) = p(y_t, g|s_t, y_{t-1}, c_t, M) + p(y_t, c|s_t, y_{t-1}, c_t, M) \]  (4)

\[
p(y_t, g|\cdot) = \begin{cases} 
\frac{1}{Z} e^{\psi_g(y_t)}, & y_t \in \mathcal{V} \\
0, & y_t \in \mathcal{X} \cap \mathcal{V} \\
\frac{1}{Z} e^{\psi_g(\text{UNK})}, & y_t \notin \mathcal{V} \cup \mathcal{X} 
\end{cases} 
\]  (5)

\[
p(y_t, c|\cdot) = \begin{cases} 
\frac{1}{Z} \sum_{j:x_j=y_t} e^{\psi_c(x_j)}, & y_t \in \mathcal{X} \\
0, & \text{otherwise} 
\end{cases} 
\]  (6)

**Generate-Mode:** The same scoring function as in the generic RNN encoder-decoder (Bahdanau et al., 2014) is used, i.e.

\[ \psi_g(y_t = v_i) = v_i^T W_o s_t, \quad v_i \in \mathcal{V} \cup \text{UNK} \]  (7)

where \( W_o \in \mathbb{R}^{(N+1) \times d_s} \) and \( v_i \) is the one-hot indicator vector for \( v_i \).

**Copy-Mode:** The score for “copying” the word \( x_j \) is calculated as

\[ \psi_c(y_t = x_j) = \sigma \left( h_j^T W_c \right) s_t, \quad x_j \in \mathcal{X} \]  (8)
CONVOLUTION NEURAL NETWORK

Next several slides borrowed from Alex Rush
Models with Sliding Windows

- Classification/prediction with sliding windows
  - E.g., neural language model
- Feature representations with sliding window
  - E.g., sequence tagging with CRFs or structured perceptron

\[
\begin{bmatrix}
  w_1 & w_2 & w_3 & w_4 & w_5
\end{bmatrix}
  w_6 & w_7 & w_8
\]

\[
\begin{bmatrix}
  w_1 & w_2 & w_3 & w_4 & w_5 & w_6
\end{bmatrix}
  w_7 & w_8
\]

\[
\begin{bmatrix}
  w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7
\end{bmatrix}
  w_8
\]

\[\vdots\]
Sliding Windows w/ Convolution

Let our input be the embeddings of the full sentence, \( X \in \mathbb{R}^{n \times d_0} \)

\[
X = [v(w_1), v(w_2), v(w_3), \ldots, v(w_n)]
\]

Define a window model as \( NN_{\text{window}} : \mathbb{R}^{1 \times (d_{\text{win}} d_0)} \mapsto \mathbb{R}^{1 \times d_{\text{hid}}} \),

\[
NN_{\text{window}}(x_{\text{win}}) = x_{\text{win}} W^1 + b^1
\]

The convolution is defined as \( NN_{\text{conv}} : \mathbb{R}^{n \times d_0} \mapsto \mathbb{R}^{(n-d_{\text{win}}+1) \times d_{\text{hid}}} \),

\[
NN_{\text{conv}}(X) = \tanh \begin{bmatrix}
NN_{\text{window}}(X_{1:d_{\text{win}}}) \\
NN_{\text{window}}(X_{2:d_{\text{win}}+1}) \\
\vdots \\
NN_{\text{window}}(X_{n-d_{\text{win}}:n})
\end{bmatrix}
\]
Pooling Operations

- Pooling “over-time” operations $f : \mathbb{R}^{n \times m} \mapsto \mathbb{R}^{1 \times m}$
  1. $f_{\text{max}}(\mathbf{X})_{1,j} = \max_i X_{i,j}$
  2. $f_{\text{min}}(\mathbf{X})_{1,j} = \min_i X_{i,j}$
  3. $f_{\text{mean}}(\mathbf{X})_{1,j} = \frac{\sum_i X_{i,j}}{n}$

$$
\begin{align*}
\downarrow & \quad \downarrow & \cdots \\
\downarrow & \quad \downarrow & \cdots \\
\downarrow & \quad \downarrow & \cdots \\
\downarrow & \quad \downarrow & \cdots \\
\downarrow & \quad \downarrow & \cdots \\
\downarrow & \quad \downarrow & \cdots \\
\downarrow & \quad \downarrow & \cdots \\
\end{align*}
= \begin{bmatrix}
\cdots \\
\cdots \\
\cdots \\
\cdots \\
\cdots \\
\cdots \\
\end{bmatrix}
$$
Convolution + Pooling

\[ \hat{y} = \text{softmax}(f_{\text{max}}(NN_{\text{conv}}(X))W^2 + b^2) \]

- \( W^2 \in \mathbb{R}^{d_{\text{hid}} \times d_{\text{out}}} \), \( b^2 \in \mathbb{R}^{1 \times d_{\text{out}}} \)
- Final linear layer \( W^2 \) uses learned window features
Multiple Convolutions

\[ \hat{y} = \text{softmax}( [f(NN_{conv}^1(X)), f(NN_{conv}^2(X)), \ldots, f(NN_{conv}^f(X))]W^2 + b^2) \]

- Concat several convolutions together.
- Each \( NN^1, NN^2 \), etc uses a different \( d_{\text{win}} \)
- Allows for different window-sizes (similar to multiple n-grams)
Convolution Diagram (Kim 2014)

- $n = 9$, $d_{\text{hid}} = 4$, $d_{\text{out}} = 2$
- red- $d_{\text{win}} = 2$, blue- $d_{\text{win}} = 3$, (ignore back channel)
## Text Classification (Kim 2014)

<table>
<thead>
<tr>
<th>Model</th>
<th>MR</th>
<th>SST-1</th>
<th>SST-2</th>
<th>Subj</th>
<th>TREC</th>
<th>CR</th>
<th>MPQA</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN-rand</td>
<td>76.1</td>
<td>45.0</td>
<td>82.7</td>
<td>89.6</td>
<td>91.2</td>
<td>79.8</td>
<td>83.4</td>
</tr>
<tr>
<td>CNN-static</td>
<td>81.0</td>
<td>45.5</td>
<td>86.8</td>
<td>93.0</td>
<td>92.8</td>
<td>84.7</td>
<td>89.6</td>
</tr>
<tr>
<td>CNN-non-static</td>
<td>81.5</td>
<td>48.0</td>
<td>87.2</td>
<td>93.4</td>
<td>93.6</td>
<td>84.3</td>
<td>89.5</td>
</tr>
<tr>
<td>CNN-multichannel</td>
<td>81.1</td>
<td>47.4</td>
<td><strong>88.1</strong></td>
<td>93.2</td>
<td>92.2</td>
<td><strong>85.0</strong></td>
<td>89.4</td>
</tr>
<tr>
<td>RAE (Socher et al., 2011)</td>
<td>77.7</td>
<td>43.2</td>
<td>82.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>86.4</td>
</tr>
<tr>
<td>MV-RNN (Socher et al., 2012)</td>
<td>79.0</td>
<td>44.4</td>
<td>82.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RNTN (Socher et al., 2013)</td>
<td>—</td>
<td>45.7</td>
<td>85.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>DCNN (Kalchbrenner et al., 2014)</td>
<td>—</td>
<td>48.5</td>
<td>86.8</td>
<td>—</td>
<td>93.0</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Paragraph-Vec (Le and Mikolov, 2014)</td>
<td>—</td>
<td><strong>48.7</strong></td>
<td>87.8</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
**AlexNet** *(krizhevsky et al., 2012)*

Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.
Discussion Points

- Strength and challenges of deep learning?

- Representation learning
  - Less efforts on feature engineering (at the cost of more hyperparameter tuning!)
  - NN learned representation today is significantly better than human engineered features
  - But the architecture that works well for image representation learning is entirely different from the architecture that works well for language

- Data
  - Most success from massive amount of clean (expensive) data
  - Recent surge of data creation type papers (especially AI challenge type tasks)
  - Which significantly limits the domains & applications
  - Need stronger models for unsupervised & distantly supervised approaches
Discussion Points

• Strength and challenges of deep learning?

• Architecture
  – allows for flexible, expressive, and creative modeling

• Easier entry to the field
  – Recent breakthrough from engineering advancements than theoretic advancements
  – Several NN platforms, code sharing culture
Transformers!
a.k.a. “Attention Is All You Need”
| SYSTEM PROMPT (HUMAN-WRITTEN) | In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.  

The scientist named the population, after their distinctive horn, Ovid’s Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.  

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. Pérez and the others then ventured further into the valley. ... |
| MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES) |
Elon Musk's scientists have announced the creation of a terrifying artificial intelligence that's so smart they refused to release it to the public.

OpenAI's GPT-2 is designed to write just like a human and is an impressive leap forward capable of penning chillingly convincing text.

It was 'trained' by analysing eight million web pages and is capable of writing large tracts based upon a 'prompt' written by a real person.

But the machine mind will not be released in its fully-fledged form because of the risk of it being used for 'malicious purposes' such as generating fake news, impersonating people online, automating the production of spam or churning out 'abusive or faked content to post on social media'.

OpenAI wrote: 'Due to our concerns about malicious applications of the technology, we are not releasing the trained model.'
Transformer models

All of these models are Transformer architecture models ... so maybe we had better learn about Transformers?

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Training Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULMfit</td>
<td>Jan 2018</td>
<td>Training</td>
<td>1 GPU day</td>
</tr>
<tr>
<td>GPT</td>
<td>June 2018</td>
<td>Training</td>
<td>240 GPU days</td>
</tr>
<tr>
<td>BERT</td>
<td>Oct 2018</td>
<td>Training</td>
<td>256 TPU days</td>
</tr>
<tr>
<td>GPT-2</td>
<td>Feb 2019</td>
<td>Training</td>
<td>~2048 TPU v3 days according to a reddit thread</td>
</tr>
</tbody>
</table>
4. The Motivation for Transformers

- We want **parallelization** but RNNs are inherently sequential

- Despite GRUs and LSTMs, RNNs still need attention mechanism to deal with long range dependencies – **path length** between states grows with sequence otherwise

- But if **attention** gives us access to any state... maybe we can just use attention and don’t need the RNN?
Transformer Overview

Attention is all you need. 2017. Aswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin

- Non-recurrent sequence-to-sequence encoder-decoder model
- Task: machine translation with parallel corpus
- Predict each translated word
- Final cost/error function is standard cross-entropy error on top of a softmax classifier

This and related figures from paper ↑
Transformer Basics

• Learning about transformers on your own?
  • Key recommended resource:
    • [http://nlp.seas.harvard.edu/2018/04/03/attention.html](http://nlp.seas.harvard.edu/2018/04/03/attention.html)
    • The Annotated Transformer by Sasha Rush
  • An Jupyter Notebook using PyTorch that explains everything!

• For now: Let’s define the basic building blocks of transformer networks: first, new attention layers!
Dot-Product Attention (Extending our previous def.)

- Inputs: a query q and a set of key-value (k-v) pairs to an output
- Query, keys, values, and output are all vectors

- Output is weighted sum of values, where
- Weight of each value is computed by an inner product of query and corresponding key
- Queries and keys have same dimensionality $d_k$ value have $d_v$

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$
Scaled Dot-Product Attention

- Problem: As $d_k$ gets large, the variance of $q^T k$ increases $\rightarrow$ some values inside the softmax get large $\rightarrow$ the softmax gets very peaked $\rightarrow$ hence its gradient gets smaller.

- Solution: Scale by length of query/key vectors:

\[
A(Q, K, V) = softmax\left(\frac{Q K^T}{\sqrt{d_k}}\right)V
\]
Scaled Dot-Product Attention

Multi-Head Attention
Self-attention in the encoder

• The input word vectors are the queries, keys and values

• In other words: the word vectors themselves select each other

• Word vector stack = Q = K = V

• We’ll see in the decoder why we separate them in the definition
Complete transformer block

- Each block has two “sublayers”
  1. Multihead attention
  2. 2-layer feed-forward NNet (with ReLU)

Each of these two steps also has:
- Residual (short-circuit) connection and LayerNorm
- Layernorm changes input to have mean 0 and variance 1, per layer and per training point (and adds two more parameters)

\[
\mu^l = \frac{1}{H} \sum_{i=1}^{H} a_i^l \\
\sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^{H} (a_i^l - \mu^l)^2}
\]

\[
h_i = f\left(\frac{g_i}{\sigma_i} (a_i - \mu_i) + b_i\right)
\]

Encoder Input

- Actual word representations are byte-pair encodings
  - As in last lecture

- Also added is a **positional encoding** so same words at different locations have different overall representations:

\[
PE_{(pos,2i)} = \sin\left(\frac{pos \cdot 10000^{2i}}{d_{model}}\right)
\]

\[
PE_{(pos,2i+1)} = \cos\left(\frac{pos \cdot 10000^{2i}}{d_{model}}\right)
\]
Complete Encoder

- For encoder, at each block, we use the same Q, K and V from the previous layer

- Blocks are repeated 6 times
  - (in vertical stack)
Attention visualization in layer 5

- Words start to pay attention to other words in sensible ways
Attention visualization: Implicit anaphora resolution

In 5th layer. Isolated attentions from just the word ‘its’ for attention heads 5 and 6. Note that the attentions are very sharp for this word.
Transformer Decoder

• 2 sublayer changes in decoder
• Masked decoder self-attention on previously generated outputs:
  ![Diagram]

• Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder
  ![Diagram]

• Blocks repeated 6 times also
  ![Diagram]
Tips and tricks of the Transformer

Details (in paper and/or later lectures):

• Byte-pair encodings
• Checkpoint averaging
• ADAM optimizer with learning rate changes
• Dropout during training at every layer just before adding residual
• Label smoothing
• Auto-regressive decoding with beam search and length penalties

• Use of transformers is spreading but they are hard to optimize and unlike LSTMs don’t usually just work out of the box and they don’t play well yet with other building blocks on tasks.
LEARNING:
BACKPROPAGATION
Inside-outside and forward-backward algorithms are just backprop.
Jason Eisner (2016).
In *EMNLP Workshop on Structured Prediction for NLP*.
Inside-Outside & Forward-Backward Algorithms are just Backprop
(tutorial paper)

Jason Eisner

“\[ \beta \uparrow \alpha \uparrow \]"
– a senior NLP researcher, in the 1990’s
Error Backpropagation

- Model parameters: \( \hat{\theta} = \{ w_{ij}^{(1)}, w_{jk}^{(2)}, w_{kl}^{(3)} \} \)

  for brevity: \( \hat{\theta} = \{ w_{ij}, w_{jk}, w_{kl} \} \)
Error Backpropagation

• Model parameters: \( \tilde{\theta} = \{w_{ij}, w_{jk}, w_{kl}\} \)

• Let \( a \) and \( z \) be the input and output of each node
Error Backpropagation

\[ z_j = g(a_j) \]

\[ a_j = \sum_i w_{ij} z_i \]
• Let \( a \) and \( z \) be the input and output of each node

\[
a_j = \sum_i w_{ij} z_i \quad a_k = \quad a_l = \quad z_j = g(a_j) \quad z_k = \quad z_l =
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]
• Let $a$ and $z$ be the input and output of each node

$$a_j = \sum_i w_{ij} z_i \quad a_k = \sum_j w_{jk} z_j \quad a_l = \sum_k w_{kl} z_k$$

$$z_j = g(a_j) \quad z_k = g(a_k) \quad z_l = g(a_l)$$
Training: minimize loss

\[ R(\theta) = \frac{1}{N} \sum_{n=0}^{N} L(y_n - f(x_n)) \]

Empirical Risk Function

\[ = \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} (y_n - f(x_n))^2 \]

\[ = \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} \left( y_n - g \left( g \left( g \left( x_{n,i} \right) \right) \right) \right)^2 \]
Training: minimize loss

\[
R(\theta) = \frac{1}{N} \sum_{n=0}^{N} L(y_n - f(x_n))
\]

Empirical Risk Function

\[
= \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} (y_n - f(x_n))^2
\]

\[
= \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} \left( y_n - g \left( \sum_k w_{kl} \left( \sum_j w_{jk} \left( \sum_i w_{ij} x_{n,i} \right) \right) \right) \right)^2
\]

Empirical Risk Function
Taking Partial Derivatives...
Error Backpropagation

Optimize last layer weights $w_{kl}$

$$L_n = \frac{1}{2} (y_n - f(x_n))^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

Calculus chain rule
Error Backpropagation

Optimize last layer weights $w_{kl}$

$L_n = \frac{1}{2} (y_n - f(x_n))^2$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial^1 (y_n - g(a_{l,n}))^2}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

Diagram:
- $x_0, x_1, x_2, \ldots, x_P$
- $z_i, a_j, z_j, a_k, z_k, a_l, z_l$
- $w_{ij}, w_{jk}, w_{kl}$$
- $f(x, \theta)$
Error Backpropagation

Optimize last layer weights $w_{kl}$

$$L_n = \frac{1}{2} (y_n - f(x_n))^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{1}{2} (y_n - g(a_{l,n}))^2 \right] \left[ \frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}} \right]$$
Error Backpropagation

Optimize last layer weights $w_{kl}$

$$L_n = \frac{1}{2} (y_n - f(x_n))^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial \frac{1}{2} (y_n - g(a_{l,n}))^2}{\partial a_{l,n}} \right] \left[ \frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}} \right] = \frac{1}{N} \sum_n \left[ -(y_n - z_{l,n}) g'(a_{l,n}) \right] z_{k,n}$$

![Diagram of a neural network with nodes and weights](image)
Error Backpropagation

Optimize last layer weights $w_{kl}$

\[ \frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right] \]

\[ \frac{\partial L_n}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}} \right] \left[ \frac{\partial z_{k,n}w_{kl}}{\partial w_{kl}} \right] = \frac{1}{N} \sum_n \left[ -(y_n - z_{l,n})g'(a_{l,n}) \right] z_{k,n} \]

Calculus chain rule
Error Backpropagation

Repeat for all previous layers

\[
\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right] = \frac{1}{N} \sum_n \left[ (y_n - z_{l,n}) g'(a_{l,n}) \right] z_{k,n} = \frac{1}{N} \sum_n \delta_{l,n} z_{k,n}
\]

\[
\frac{\partial R}{\partial w_{jk}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{k,n}} \right] \left[ \frac{\partial a_{k,n}}{\partial w_{jk}} \right] = \frac{1}{N} \sum_n \left[ \sum_l \delta_{l,n} w_{kl} g'(a_{k,n}) \right] z_{j,n} = \frac{1}{N} \sum_n \delta_{k,n} z_{j,n}
\]

\[
\frac{\partial R}{\partial w_{ij}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{j,n}} \right] \left[ \frac{\partial a_{j,n}}{\partial w_{ij}} \right] = \frac{1}{N} \sum_n \left[ \sum_k \delta_{k,n} w_{jk} g'(a_{j,n}) \right] z_{i,n} = \frac{1}{N} \sum_n \delta_{j,n} z_{i,n}
\]
Backprop Recursion

\[ z_j = g(a_j) \]

\[ a_j = \sum_i w_{ij} z_i \]

\[ \frac{\partial R}{\partial w_{jk}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_k} \right] \left[ \frac{\partial a_k}{\partial w_{jk}} \right] = \frac{1}{N} \sum_n \left[ \sum_l \delta_{l,n} w_{kl} g'(a_k) \right] z_j, n = \frac{1}{N} \sum_n \delta_{k,n} z_{j,n} \]

\[ \frac{\partial R}{\partial w_{ij}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_j} \right] \left[ \frac{\partial a_j}{\partial w_{ij}} \right] = \frac{1}{N} \sum_n \left[ \sum_k \delta_{k,n} w_{jk} g'(a_j) \right] z_i, n = \frac{1}{N} \sum_n \delta_{j,n} z_{i,n} \]
Learning: Gradient Descent

\[
\begin{align*}
    w_{ij}^{t+1} &= w_{ij}^t - \eta \frac{\partial R}{w_{ij}} \\
    w_{jk}^{t+1} &= w_{jk}^t - \eta \frac{\partial R}{w_{kl}} \\
    w_{kl}^{t+1} &= w_{kl}^t - \eta \frac{\partial R}{w_{kl}}
\end{align*}
\]
Backpropagation

- Starts with a forward sweep to compute all the intermediate function values.
- Through backprop, computes the partial derivatives recursively.
- A form of dynamic programming.
  - Instead of considering exponentially many paths between a weight $w_{ij}$ and the final loss (risk), store and reuse intermediate results.
- A type of automatic differentiation. (there are other variants e.g., recursive differentiation only through forward propagation.)
Backpropagation

- TensorFlow (https://www.tensorflow.org/)
- Torch (http://torch.ch/)
- Theano (http://deeplearning.net/software/theano/)
- CNTK (https://github.com/Microsoft/CNTK)
- cnn (https://github.com/clab/cnn)
- Caffe (http://caffe.berkeleyvision.org/)

Primary Interface Languages:

- Python
- Lua
- Python
- C++
- C++
- C++

Forward Gradient
Cross Entropy Loss (aka log loss, logistic loss)

- Cross Entropy
  \[ H(p, q) = - \sum_y p(y) \log q(y) \]

- Related quantities
  \[ H(p) = \sum_y p(y) \log p(y) \]
  - Entropy
  - KL divergence (the distance between two distributions p and q)
    \[ D_{KL}(p||q) = \sum_y p(y) \log \frac{p(y)}{q(y)} \]
    \[ H(p, q) = E_p[-\log q] = H(p) + D_{KL}(p||q) \]

- Use Cross Entropy for models that should have more probabilistic flavor (e.g., language models)
- Use Mean Squared Error loss for models that focus on correct/incorrect predictions
  \[ \text{MSE} = \frac{1}{2} (y - f(x))^2 \]
NEURAL CHECK LIST
Neural Checklist Models
(Kiddon et al., 2016)

• What can we do with gating & attention?
Encoder--Decoder Architecture

Chop
<\(s\)>
Chop
the
tomatoes
.
Add

garlic tomato salsa

Doesn't address changing ingredients

Want to update ingredient information as ingredients are used
Encode title - decode recipe

Cut each sandwich in halves.
Sandwiches with sandwiches, sandwiches, sandwiches, sandwiches, sandwiches, sandwiches, sandwiches, sandwiches, sandwiches or sandwiches or triangles, a griddle, each sandwich.
Top each with a slice of cheese, tomato, and cheese.
Top with remaining cheese mixture.
Top with remaining cheese.
Broil until tops are bubbly and cheese is melted, about 5 minutes.
Recipe generation vs machine translation

Two input sources

• Only ~6-10% words align between input and output.
• The rest must be generated from context (and implicit knowledge about cooking)
• Contextual switch between two different input sources
Chop tomatoes. Add Chop tomatoes.

<\s> Chop the tomatoes. Add
garlic tomato salsa

Doesn’t address changing ingredients

Want to update ingredient information as ingredients are used
Neural checklist model

**Key**
- gate
- sum
- sigmoid
- linear projection
- multiplication
- softmax
- linear interpolation
- select dimension

**Update checklist**
- \( E_{t+1}^{\text{new}} \)
- \( E_{t+1}^{\text{used}} \)
- \( a_t \)
- \( 1-a_t \)

**Update available and used agenda items**
- \( X \)
- \( E \)
- \( a_t \)
- \( 1-a_t \)

**Attention mechanisms**
- \( X \)
- \( \sigma \)
- \( h_t \)
- \( z_t \)
- \( s_t \)
- \( q_t \)
- \( r_t \)

**GRU language model**
- \( h_{t-1} \)
- \( x_t \)
- \( g \)

**Generate output**
- \( o_t \)
- \( f_t \)
- \( Ph_t \)
- \( E_{t+1}^{\text{new}} \)
- \( E_{t+1}^{\text{used}} \)

**Hidden state projected into agenda space**
- \( \text{ref-type}(h_t) \)

**Probability of using new item**
- \( f_t^{\text{new}} \)
Let’s make salsa!

Garlic tomato salsa

tomatoes
onions
garlic
salt
Neural checklist model

hidden state classifier:
- non-ingredient
- new ingredient
- used ingredient

which ingredients are still available

LM

new hidden state

Chop

<\text{S}>

garlic  tomato  salsa
Chop

the

tomatoes

0.85

0.10

0.04

0.01

new
ingredient

non-ingredient

Chop

the

tomatoes

✓
Neural checklist model

Dice

the

onions

0.00
0.94
0.03
0.01

✓

✓

✓
Neural checklist model

Add to tomatoes

0.94
0.04
0.01
0.01

used ingredient

✓✓✓

✓✓✓
Checklist is probabilistic

\[ \alpha_t^{new} = \text{new ingredient prob. distribution} \]
\[ a_t^{new} = P(\bullet | h_t) \cdot \alpha_t^{new} \]
\[ a_{t+1} = a_t + a_t^{new} \]
Hidden state classifier is soft

Add tomatoes to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.

Add to tomatoes.
Interpolation

\[ W_o \in \mathbb{R}^{|V| \times k} \]

\[ w_t = \text{softmax}(W_o h_t) \]

\[ o_t = P(\text{gray} | h_t) c_t^{LM} \]
\[ + P(\text{red} | h_t) c_t^{new} \]
\[ + P(\text{blue} | h_t) c_t^{used} \]

\[ c_t^{LM} = W_h h_t \]

\[ W_h \in \mathbb{R}^{k \times k} \]

Attention model over available ingredients

Attention model over used ingredients

probability distribution over vocabulary
Choose ingredient via attention

\[ \alpha_t^{new} = \text{softmax}(\gamma E_t^{new} c_t^{LM}) \]

- **Temperature term**
- **Content vector from language model**
- **Available ingredient embeddings**

**Attention models for other NLP tasks**
- MT (Balasubramanian et al. 13, Bahdanau et al. 14)
- Sentence summarization (Rush et al. 15)
- Machine reading (Cheng et al. 16)
- Image captioning (Xu et al. 15)
Attention-generated embeddings

Can generate an embedding from the attention probabilities

\[ c_t^{\text{new}} = E^T \alpha_t^{\text{new}} \]
Cook eggplant in boiling water, covered, for 10 min. Drain and cut in half lengthwise. Scoop out insides leaving 1/2 " shell. Mash insides with cottage cheese, onion, bay leaf, basil, oregano and tomato sauce. Preheat oven to 350 °C; stuff eggplant halves, place in casserole dish and bake covered for 15 min. Add a little water to bottom of pan to keep eggplant moist. Top with provolone cheese. Bake 5 more min uncovered. 1 serving =

In a small bowl, combine the cheese, eggplant, basil, oregano, tomato sauce and onion. Mix well. Shape mixture into 6 patties, each about 3/4-inch thick. Place on baking sheet. Bake at 350 degrees for 30 minutes or until lightly browned. 

Southern living magazine, sometime in 1980. Typed for you by Nancy Coleman.