Computational Cognitive Modeling

Neural networks and deep learning
(part 2)

Brenden Lake & Todd Gureckis

email address for instructors:
instructors-ccm-spring2019@nyuccl.org

course website:
https://brendenlake.github.io/CCM-site/
Two very important types of neural networks

- **Recurrent neural network**
  - Output units
  - Hidden units
  - Input units
  - Context units

- **Deep convolutional neural network**
  - Layers of feature maps

Input: a daisy image
Output: “daisy”
Data often has important temporal structure

- Language/text
- Speech
- Video
- Financial
- Weather
- All of human behavior unfolds in time
How do we represent time in a neural network?

Naive approach: represent time spatially in a standard network

Problems with this approach:

• The network has a fixed buffer. How large do you make the buffer?

• Two identical patterns translated in time, such as \([0 \ 1 \ 1 \ 0 \ 0 \ 0]\), \([0 \ 0 \ 0 \ 1 \ 1 \ 0]\) have no natural overlap in the (untrained) architecture.
Finding Structure in Time

JEFFREY L. ELMAN
University of California, San Diego

Time underlies many interesting human behaviors. Thus, the question of how to represent time in connectionist models is very important. One approach is to represent time implicitly by its effects on processing rather than explicitly (as in a spatial representation). The current report develops a proposal along these lines first described by Jordan (1986) which involves the use of recurrent links in order to provide networks with a dynamic memory. In this approach, hidden unit patterns are fed back to themselves; the internal representations which develop thus reflect task demands in the context of prior internal states. A set of simulations is reported which range from relatively simple problems (temporal version of XOR) to discovering syntactic/semantic features for words. The networks are able to learn interesting internal representations which incorporate task demands with memory demands; indeed, in this approach the notion of memory is inextricably bound up with task processing. These representations reveal a rich structure, which allows them to be highly context-dependent, while also expressing generalizations across classes of items. These representations suggest a method for representing lexical categories and the type/token distinction.

INTRODUCTION

Time is clearly important in cognition. It is inextricably bound up with many behaviors (such as language) which express themselves as temporal sequences. Indeed, it is difficult to know how one might deal with such basic problems as goal-directed behavior, planning, or causation without some way of representing time.

The question of how to represent time might seem to arise as a special problem unique to parallel-processing models, if only because the parallel nature of computation appears to be at odds with the serial nature of tem-
Simple recurrent network (SRN; or Elman network)
A simple example task

input data:
diibaguuubadiidiiguuuuguuuudiidiibadii....
(random mix of "ba", "dii", and "guu")
task:
predict the next letter

representation of input

<table>
<thead>
<tr>
<th></th>
<th>Consonant</th>
<th>Vowel</th>
<th>Interrupted</th>
<th>High</th>
<th>Back</th>
<th>Voiced</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>[1 0 1 0 0 1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>[1 0 1 1 0 1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>[1 0 1 0 1 1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>[0 1 0 0 1 1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>[0 1 0 1 0 1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>[0 1 0 1 1 1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance of the trained network

Error (root mean squared error)

Network shows higher error when predicting unpredictable characters (b, d, g)

Words “ba”, “di”, and “guu”
What is a “word”? 

A similar simulations with a sequence of artificial sentences such as “many years ago a boy and girl lived by the sea..”

- Much previous work in cognitive science and linguistics assumed basic linguistic units such as phonemes, morphemes, words, etc.
- But the definition of a “word” isn’t that clear
- How many words is each of these phrases? (examples from PDP Handbook)
- Elman’s paper and the SRN were the first to break away from commitments to basic linguistic units (phonemes, morphemes, words, etc.)
Training a recurrent neural network

notation for SRN

output

\[ \hat{y}_t \]

\[ h_{t-1} \]

\[ h_t \]

input

\[ x_t \]

weight matrices \( V, W, U \)

index \( t \) is represents tie step
Unrolling a recurrent network in time

Global error \[ E = \sum_t E_t \]

Error terms

\hline
<table>
<thead>
<tr>
<th>E0</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{y}_0 )</td>
<td>( \hat{y}_1 )</td>
<td>( \hat{y}_2 )</td>
<td>( \hat{y}_3 )</td>
</tr>
<tr>
<td>( h_0 )</td>
<td>( h_1 )</td>
<td>( h_2 )</td>
<td>( h_3 )</td>
</tr>
<tr>
<td>( x_0 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>( x_3 )</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>

weight matrices V, W, U
Unrolling a recurrent network in time

Global error \[ E = \sum_t E_t \]

Error terms

\[ E_0 \]
\[ \hat{y}_0 \]
\[ h_{-1} \rightarrow h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \]
\[ x_0 \rightarrow i \rightarrow i \rightarrow i \rightarrow i \]
\[ v \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow v \]
\[ w \rightarrow \leftarrow \leftarrow \leftarrow \leftarrow w \]

Input sequence: diibaguuu

weight matrices V, W, U
Reminder: Backpropagation algorithm for computing gradient

Multi-step strategy:

\[
\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial h} \frac{\partial h}{\partial w_3}
\]

Step 1) Compute how error changes as a function of hidden unit activation

Step 2) Compute how hidden unit activation changes as a function of weight

The cost function is:

\[
E(w, b) = (\hat{y} - y)^2 = (g(\text{net}) - y)^2
\]
Backpropagation through time

Global error

\[ E = \sum_t E_t \quad \frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W} \]

Illustration for computing \( \frac{\partial E_2}{\partial W} \)

Error terms

E₀ \quad \hat{y}_0 \quad \text{U} \quad \text{V} \quad \text{W}

E₁ \quad \hat{y}_1 \quad \text{U} \quad \text{V} \quad \text{W}

E₂ \quad \hat{y}_2 \quad \frac{\partial E_2}{\partial h_2} \quad \text{U} \quad \text{V} \quad \text{W} \quad \text{W}

E₃ \quad \hat{y}_3 \quad \text{U} \quad \text{V} \quad \text{W}

Backprop step 1

Weight matrices V, W, U
Backpropagation through time

Global error

\[ E = \sum_t E_t \quad \frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W} \]

Illustration for computing \( \frac{\partial E_2}{\partial W} \)

Error terms

\( E_0 \)

\( \hat{y}_0 \)

\( U \)

\( \text{h}_0 \)

\( x_0 \)

\( E_1 \)

\( \hat{y}_1 \)

\( U \)

\( \text{h}_1 \)

\( \text{v} \)

\( \text{x}_1 \)

\( \frac{\partial h_2}{\partial h_1} \)

\( \frac{\partial h_1}{\partial W} \)

\( \text{h}_2 \)

\( \text{x}_2 \)

\( \frac{\partial E_2}{\partial h_2} \)

\( \text{h}_3 \)

\( \text{x}_3 \)

\( \text{h}_1 \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)

\( \text{v} \)
Backpropagation through time

Global error

\[ E = \sum_t E_t \quad \frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W} \]

Illustration for computing \( \frac{\partial E_2}{\partial W} \)

Error terms

\( E_0 \)

\( \hat{y}_0 \)

\( \frac{\partial h_1}{\partial h_0} \)

\( \frac{\partial h_0}{\partial W} \)

\( x_0 \)

\( h_0 \)

\( U \)

\( V \)

\( W \)

\( h_1 \)

\( \hat{y}_1 \)

\( \frac{\partial h_2}{\partial h_1} \)

\( \frac{\partial h_1}{\partial W} \)

\( x_1 \)

\( h_2 \)

\( \hat{y}_2 \)

\( \frac{\partial E_2}{\partial h_2} \)

\( \frac{\partial h_2}{\partial h_1} \)

\( \frac{\partial h_1}{\partial W} \)

\( x_2 \)

\( h_3 \)

\( \hat{y}_3 \)

\( \frac{\partial h_3}{\partial h_2} \)

\( \frac{\partial h_2}{\partial W} \)

\( x_3 \)

backprop step 3

weight matrices \( V, W, U \)

Now, we sum the gradients from the last three slides.

And compute gradients for other time steps, \( E_0, E_1 \) again summing over shared weights...

Conceptually, still no different than wiggling \( W \) and seeing how error changes!
Fortunately, PyTorch (or TensorFlow, etc.) can compute all of the gradients for you!

```python
class SRN(nn.Module):
    def __init__(self, nsymbols, hidden_size):
        # nsymbols: number of possible input/output symbols
        super(SRN, self).__init__()
        self.hidden_size = hidden_size
        self.12h = nn.Linear(nsymbols + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, nsymbols)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        # input: [1 x nsymbol tensor] with one-hot encoding of a single symbol
        # hidden: [1 x hidden size tensor] which is the previous hidden state
        combined = torch.cat((input, hidden), 1)  # tensor size 1 x (nsymbol + 1)
        hidden = self.12h(combined)  # tensor size 1 x hidden size
        hidden = F.sigmoid(hidden)
        output = self.h2o(hidden)  # 1 x nsymbol
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size, requires_grad=True)

def train(seq_tensor, rnn):
    # seq_tensor: [seq_length x 1 x nsymbols tensor]
    # rnn: instance of SRN class
    hidden = rnn.initHidden()
    rnn.train()
    rnn.zero_grad()
    loss = 0
    seq_length = seq_tensor.shape[0]
    for i in range(seq_length-1):
        output, hidden = rnn(seq_tensor[i], hidden)
        loss += criterion(output, variableToIndex(seq_tensor[i+1]))
    loss.backward()
    optimizer.step()
    return loss.item() / float(seq_length-1)
```

(this line computes all of the gradients for you.)
Discovering lexical classes from simple sentences

(also Elman, 1990)

"man eat cookie"
"woman see book"
"dragon eat human"
...

Can the SRN discover lexical classes like nouns and verbs?
Discovering lexical classes from simple sentences

Template for generating simple sentences

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOUN-HUM</td>
<td>man, woman</td>
</tr>
<tr>
<td>NOUN-ANIM</td>
<td>cat, mouse</td>
</tr>
<tr>
<td>NOUN-INANIM</td>
<td>book, rock</td>
</tr>
<tr>
<td>NOUN-AGRESS</td>
<td>dragon, monster</td>
</tr>
<tr>
<td>NOUN-FRAG</td>
<td>glass, plate</td>
</tr>
<tr>
<td>NOUN-FOOD</td>
<td>cookie, break</td>
</tr>
<tr>
<td>VERB-INTRAN</td>
<td>think, sleep</td>
</tr>
<tr>
<td>VERB-TRAN</td>
<td>see, chase</td>
</tr>
<tr>
<td>VERB-AGPAT</td>
<td>move, break</td>
</tr>
<tr>
<td>VERB-PERCEPT</td>
<td>smell, see</td>
</tr>
<tr>
<td>VERB-DESTROY</td>
<td>break, smash</td>
</tr>
<tr>
<td>VERB-EAT</td>
<td>eat</td>
</tr>
</tbody>
</table>

“man eat cookie”
“woman see book”
“dragon eat human”
...

<table>
<thead>
<tr>
<th>Templates for Sentence Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD 1</td>
</tr>
<tr>
<td>NOUN-HUM</td>
</tr>
<tr>
<td>NOUN-HUM</td>
</tr>
<tr>
<td>NOUN-HUM</td>
</tr>
<tr>
<td>NOUN-HUM</td>
</tr>
<tr>
<td>NOUN-HUM</td>
</tr>
<tr>
<td>NOUN-HUM</td>
</tr>
<tr>
<td>NOUN-ANIM</td>
</tr>
<tr>
<td>NOUN-ANIM</td>
</tr>
<tr>
<td>NOUN-ANIM</td>
</tr>
<tr>
<td>NOUN-ANIM</td>
</tr>
<tr>
<td>NOUN-ANIM</td>
</tr>
<tr>
<td>NOUN-INANIM</td>
</tr>
<tr>
<td>NOUN-AGRESS</td>
</tr>
<tr>
<td>NOUN-AGRESS</td>
</tr>
<tr>
<td>NOUN-AGRESS</td>
</tr>
<tr>
<td>NOUN-AGRESS</td>
</tr>
<tr>
<td>NOUN-AGRESS</td>
</tr>
</tbody>
</table>

| WORD 2                          |
| NOUN-HUM                        |
| VERB-EAT                        |
| VERB-DESTROY                    |
| NOUN-HUM                        |
| NOUN-FOOD                       |
| NOUN-ANANIM                     |
| NOUN-AGRESS                     |
| NOUN-AGRESS                     |
| NOUN-AGRESS                     |
| NOUN-HUM                        |
| VERB-AGPAT                      |
| VERB-AGPAT                      |
| VERB-AGPAT                      |
| VERB-AGPAT                      |

| WORD 3                          |
| NOUN-FOOD                       |
| NOUN-INANIM                     |
| NOUN-ANANIM                     |
| NOUN-AGRESS                     |
| NOUN-AGRESS                     |
| NOUN-HUM                        |
| NOUN-ANANIM                     |
Discovering lexical classes from simple sentences

"one-hot" encoding for words

TABLE 5
Fragment of Training Sequences for Sentence Simulation

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000000000000000000100 (woman)</td>
<td>000000000000000000000000000000010000 (smash)</td>
</tr>
<tr>
<td>00000000000000000000000000000010000 (smash)</td>
<td>00000000000000000000000000000001000000000 (plate)</td>
</tr>
<tr>
<td>000000000000000000000000000000100000000 (plate)</td>
<td>0000010000000000000000000000000001 (cat)</td>
</tr>
<tr>
<td>000000000000000000000000000000001000000000 (cat)</td>
<td>0000000000000000000000000000000001000000000 (move)</td>
</tr>
<tr>
<td>00000000000000000000000000000000001000000000 (move)</td>
<td>0000000000000000000000000000000001000000000 (man)</td>
</tr>
<tr>
<td>00000000000000000000000000000000000000000001000000000 (man)</td>
<td>00001000000000000000000000000000000100000000000 (break)</td>
</tr>
<tr>
<td>0001000000000000000000000000000000000000000000000 (break)</td>
<td>000100000000000000000000000000000000000000 (car)</td>
</tr>
<tr>
<td>0000100000000000000000000000000000000000000000000 (car)</td>
<td>010000000000000000000000000000000000000000 (boy)</td>
</tr>
<tr>
<td>0100000000000000000000000000000000000000000000 (boy)</td>
<td>000000000000000000000000000000000000000000 (move)</td>
</tr>
<tr>
<td>0000000000000000000000000000000000000000000000000 (move)</td>
<td>000000000000000000000000000000000000000000000 (girl)</td>
</tr>
<tr>
<td>0000000000000000000000000000000000000000000000000 (girl)</td>
<td>000000000000000000000000000000000000000000 (eat)</td>
</tr>
<tr>
<td>0000000000001000000000000000000000000000000000000 (eat)</td>
<td>010000000000000000000000000000000000000000 (bread)</td>
</tr>
<tr>
<td>0010000000000000000000000000000000000000000000000 (bread)</td>
<td>000000000010000000000000000000000000000000 (dog)</td>
</tr>
<tr>
<td>0000000000000000000000000000000000000000000000000 (dog)</td>
<td>000000000000000000000000000000000000000000 (move)</td>
</tr>
<tr>
<td>0000000000000000000000000000000000000000000000000 (move)</td>
<td>000000000000000000000000000000000000000000 (mouse)</td>
</tr>
<tr>
<td>0000000000000000000000000000000000000000000000000 (mouse)</td>
<td>000000000000000000000000000000000000000000 (move)</td>
</tr>
<tr>
<td>0000000000000000000000000000000000000000000000000 (move)</td>
<td>010000000000000000000000000000000000000000 (book)</td>
</tr>
<tr>
<td>0000000000000000000000000000000000000000000000000 (book)</td>
<td>000000000000000000000000000000000000000000 (lion)</td>
</tr>
</tbody>
</table>
The network just learned to predict the next word from the previous word.

This diagram shows results of clustering the average pattern over the hidden units for each word (across many presentations).

From just the prediction task, the network “discovers” nouns vs. verbs, and also animate vs inanimate, etc. without building in these lexical classes in any way.

Discovering lexical classes from simple sentences
It’s a relatively small step to state-of-the-art recurrent neural networks for text processing and machine translation.
Long short-term memory (LSTM)

- Simple recurrent networks (SRNs) are not used very often anymore. Instead, they have been replaced by more powerful Long short term memory (LSTM) recurrent networks (Hochreiter & Schmidhuber, 1997).
- Conceptually, LSTMs are recurrent networks just like SRNs, except the “hidden state” can more easily pass from one step to another without modification, unless the network decides to modify it. This gives the network a much longer memory.
- For state-of-the-art text modeling, networks are trained to predict the next word given the previous, just as we studied in the previous examples. Training is with backprop through time.
Long short-term memory (LSTM)

Important notes:
- all vectors including cell, hidden, forget, etc. have same dimensionality (except $x$)
- biases not shown in equations for simplicity

 updating the cell state
forget some cell elements $f$, add to other elements $i$
\[ c_t = f_t c_{(t-1)} + i_t g_t \]

forget gate
compute which cell elements to forget ($f$)
\[ f_t = \text{logistic}(W_{if}x_t + W_{hf}h_{(t-1)}) \]

input gate
compute which cell elements to add to ($i$), and what to add ($g$)
\[ i_t = \text{logistic}(W_{ii}x_t + W_{hi}h_{(t-1)}) \]
\[ g_t = \tanh(W_{ig}x_t + W_{hg}h_{(t-1)}) \]

output gate
compute which cell elements to output ($o$), and output them ($h$)
\[ o_t = \text{logistic}(W_{io}x_t + W_{ho}h_{(t-1)}) \]
\[ h_t = o_t \tanh(c_t) \]
Neural machine translation with recurrent neural networks

- Recurrent neural networks can solve sequence-to-sequence (seq2seq) problems by having a recurrent encoder, which embeds a sentence in the hidden space, and another recurrent decoder that translates it.
- This is the main technical juice behind Google Translate!
- Trained using sentence pairs and backprop through time

(e.g., Sutskever, Vinyals, & Le, 2014)
Critiques of recurrent neural networks
(Marcus, 1998, Rethinking eliminative connectionism)

- Recurrent neural networks (RNNs) generalize very well WITHIN a specific set of words / symbols.
- However, they generalize very poorly to new words OUTSIDE the training set.
- Say the network is trained on sentences such as, “a rose is a rose”, “a daisy is a daisy”, and “a violet is a violet”.
- It will NOT generalize to a new word, “a blicket is a blicket”.
- People can easily learn rules that apply to arbitrary new variables.

Sentence input... “A rose is a”
Today’s lecture: Two very important types of neural network models

Recurrent neural network
- Output units
- Hidden units
- Input units
- Context units

Deep convolutional neural network
- Layers of feature maps
- Input
- Output: “Daisy”
convolving an image with a filter

We slide the filter across the image to produce an output image.
Deep convolutional neural network (convnet) for vision

\[ g(\text{net}_i) = \frac{e^{\text{net}_i}}{\sum_c e^{\text{net}_c}} \]

output classes (c) 0 1 2... 9

Filter bank + non-linearity

Max pooling

Layer 2: 12 feature maps
Filter bank + non-linearity

Max pooling

Layer 1: 4 feature maps
Filter bank + non-linearity

example convolution

Slide courtesy of Yann LeCun
Deep convolutional neural network

Learned filters in a deep convnet

• Key assumption of “translation invariance”: If a filter (e.g., a horizontal edge detector) is useful in one part of the image, it is probably useful anywhere in the image.

(some filters learned from Krizhevsky et al., 2012)
Let’s go through computing with convolutions to build a feature map...
Each rectangular image is a feature map applied to the image of a Samoyed dog (bottom left; and RGB (red, green, blue) inputs, bottom right). Each rectangular image is a feature map applied to the image of a Samoyed dog (bottom left; and RGB (red, green, blue) inputs, bottom right).
The backpropagation procedure to compute the gradient of an empirical loss with respect to the weights of a network is at the heart of deep learning. The backpropagation equation can be applied repeatedly to propagate gradients through all modules, starting from the output at the top (where the network produces its prediction) all the way to the input or output layer are conventionally called hidden units. The inputs of the first layer are conventionally called the input volume, while the outputs of the last layer are conventionally called the output volume.

To make classifiers more powerful, one can use generic non-linear combinations of channel responses, such as linear combinations of weighted sums of channels. These combinations can be computed by considering several channels to be independent random variables with a common distribution and then computing the linear combinations of these variables. This allows for the efficient computation of linear combinations of channels, which can then be used to train multilayer architectures.

The backpropagation procedure is used to train multilayer architectures by computing the gradient of the empirical loss with respect to the weights of each layer. This is done by first computing the gradient of the loss with respect to the output of the network, and then backpropagating this gradient through all layers to compute the gradients with respect to the weights of each layer. This allows the weights to be updated to minimize the loss, and the network to be trained to classify images with high accuracy.

In practice, poor local minima are rarely a problem with large networks. Regardless of the initial conditions, the system nearly always reaches solutions of very similar quality. Recent theoretical and empirical results strongly suggest that local minima are not a serious obstacle to learning.

In general, the most popular non-linear function is the rectified linear unit (ReLU) function, defined as \( f(z) = \max(z, 0) \). The ReLU function is advantageous because it is computationally inexpensive and has several useful properties, such as being differentiable everywhere and having a simple closed-form expression for the derivative.

The ReLU function is often used in deep neural networks because it can help to prevent the vanishing gradient problem, which can occur when using other non-linear functions like the sigmoid or tanh functions. The ReLU function also has the property of being sparse, which means that it tends to activate only a small number of units at a time. This can help to improve the interpretability of the network by making it easier to understand which features are being learned and how they are being combined.

In conclusion, the backpropagation procedure is a key component of deep learning, allowing for the training of powerful multilayer architectures by computing gradients with respect to the weights of each layer. The most popular non-linear function used in deep learning is the ReLU function, which is advantageous due to its simplicity, efficiency, and ability to help prevent the vanishing gradient problem.
Each rectangular image is a feature map applied to the image of a Samoyed dog (bottom left; and RGB (red, green, blue) inputs, bottom right). Each module in the stack transforms its input to increase both the selectivity and the ability for each of several categories. To go from one layer to the next, one computes the non-linear input–output mapping. The key insight is that the derivative (or gradient) of the surface curves up in most dimensions and curves down in the dimension corresponding to the output. A deep-learning architecture is a multilayer stack of simple modules (for example, an image) to a fixed-size output (for example, a probability distribution over object classes). The conventional option is the Gaussian kernel, but generic features such as those arising with the Gaussian kernel do not allow the learner to generalize well. A deep network acts as a non-linear transformation of the input. To train such a network, one can define an objective that rewarded the performance of the network on the training data, and then compute non-linear input–output mappings. Each module in the network architectures (Fig. 1), which learn to map a fixed-size input (for example, an image) to a fixed-size output (for example, a probability distribution over object classes). The backpropagation procedure to compute the gradient of an empirical result strongly suggests that local minima are not a serious issue in general. Instead, the landscape is packed with a combination of minima, maxima, and saddle points. The backpropagation procedure to compute the gradient of an empirical result strongly suggests that local minima are not a serious issue in general. Instead, the landscape is packed with a combination of minima, maxima, and saddle points.
The outputs (not the filters) of modules are nothing more than a practical application of the chain rule for derivatives. The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be computed by working backwards from the gradient with respect to the output of that module (or the input of the subsequent module). Once these gradients have been computed, it is straightforward to compute the gradients with respect to the weights of each module.

As long as the modules are relatively smooth functions of their inputs and of their internal weights, one can compute gradients using the backpropagation equation, which can be applied repeatedly to propagate gradients through all modules, starting from the output at the top (where the network produces its prediction) all the way to the output of the lowest module (or the input of the subsequent lowest module). The backpropagation equation is:

\[

g(z) = \begin{cases} 
1 & \text{for } z > 0 \\
0 & \text{for } z \leq 0 
\end{cases}
\]

where \( g(z) \) is the derivative of the activation function with respect to its input. The gradient of the objective function with respect to the weights of a multilayer stack of modules during the 1970s and 1980s that it worked, was discovered independently by several different works.

In networks with many layers, the surface curves up in most dimensions and curves down in the one that leads to the best solution, which is a very flat minimum. The conventional option is to hand design good feature extractors, which requires a considerable amount of work.
Let’s skip to compute the next feature map...
Channel 0 (e.g., red)
Channel 1 (e.g., green)
Channel 2 (e.g., blue)

Filter 0
Filter 1

Next layer of network
Feature map 0
Feature map 1

Input Volume (+pad 1) (7x7x3)
x[:, :, 0]
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0

Filter W0 (3x3x3)
w0[:, :, 0]
0 1 0
-1 1 0
0 1 1

Filter W1 (3x3x3)
w1[:, :, 0]
0 1 1
-1 -1 1
0 1 1

Output Volume (3x3x2)
o[:, :, 0]
8 6 5
7 9 2
-1 -7 3

Bias b0 (1x1x1)
b0[:, :, 0]
1

Bias b1 (1x1x1)
b1[:, :, 0]
0

Feature maps
Convolutions and channels
Backpropagation to train multilayer architectures.

A deep-learning architecture is a multilayer stack of simple modules, all (or most) of which are subject to learning, and many of which compute non-linear input–output mappings. Each module in the network architectures (Fig. 1), which learn to map a fixed-size input (for example, an image) to a fixed-size output (for example, a probability vector over classes).

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

As long as the modules are relatively smooth functions of their inputs and of their internal weights, one can compute gradients using the backpropagation procedure. The idea that this could be done, and that it worked, was discovered independently by several different works. Regardless of the initial conditions, the system nearly always reaches solutions of very similar quality. Recent theoretical and empirical work (Fig. 2) shows that the resilience to poor local minima is partly large number of saddle points where the gradient is zero, and the system is likely to converge to solutions of very similar quality. Recent theoretical and empirical work (Fig. 2) shows that the resilience to poor local minima is partly due to the fact that the training process is likely to converge to solutions of very similar quality.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.

The backpropagation procedure to compute the gradient of an objective with respect to the input of a module can be implemented by working backwards from the gradient with respect to the output of the network (Fig. 2). The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be expressed using the chain rule for derivatives. This is why shallow classifiers can be trained in a forward pass of working from input to output, but deep networks require a backpropagation procedure to work from output to input and compute gradients for the weights.
After convolutions, apply a non-linear activation function (e.g., ReLUs)

\[
g(\text{net}) = \begin{cases} 
\text{net} & \text{net} \geq 0 \\
0 & \text{net} < 0 
\end{cases}
\]

Feature map 0

<table>
<thead>
<tr>
<th>8</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>-1</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

Feature map 1

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>-5</td>
<td>-1</td>
<td>-2</td>
</tr>
</tbody>
</table>

Rectified linear unit (ReLU)

\[
g(\text{net})
\]

\[
\begin{array}{ccc}
8 & 6 & 5 \\
7 & 9 & 2 \\
0 & 7 & 3 \\
\end{array}
\]

\[
\begin{array}{ccc}
2 & 3 & 7 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\end{array}
\]
Next step: Pooling

- Downsamples a feature map to a coarser resolution
- Provides additional translation invariance

Max pooling

<table>
<thead>
<tr>
<th>Single depth slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Max pool with 2x2 filters and stride 2

6 8
3 4

Max pooling

Convolutions and ReLU

Red  Green  Blue
Connection to biological architecture of primary visual cortex
(Hubel & Wiesel)

stimulus

simple cell

spikes

Stimulus: on off

localized edge detector

complex cell

Stimulus: on off

invariance through pooling

Max pooling

Convolutions and ReLU

Red

Green

Blue
Deep convolutional neural network

Samoyed (16); Papillon (5.7); Pomeranian (2.7); Arctic fox (1.0); Eskimo dog (0.6); white wolf (0.4); Siberian husky (0.4)

How do we train it? Backpropagation

Training data (ImageNet)
- 1.2 million images
- 1000 categories
- ~1200 images per category

**AlexNet**
8 layers
~60 million parameters

**GoogLeNet**
22 layers
~5 million parameters

**Residual network**
152 layers
Example results on test images

object recognition

finding similar images

(probe similar images...)

(some filters learned by Krizhevsky et al., 2012)
Deep convnets for understanding psychological and neural representations

- Peterson, Abbott, and Griffiths (2016) explored convnets for predicting similarity ratings between images.
- Lake, Zaremba, Fergus, and Gureckis (2015) showed that deep convnets can predict human typicality ratings for some classes of natural images.
- Yamins et al. (2014) showed that deep convnets can predict neural response in high-level visual areas.
Some images look more similar to us than others. Can a neural network trained for classification help to explain similarity judgments from humans?

example animal images (they collected 120 x 120 pairwise ratings)
computing image-to-image similarity

\[ \text{sim}(i, j) = \sum_k f_{ik} f_{jk} \] 

similarity computed as dot product

summarizing images as high-level feature vector \( f \)

Samoyed (16); Papillon (5.7); Pomeranian (2.7); Arctic fox (1.0); Eskimo dog (0.6); white wolf (0.4); Siberian husky (0.4)
Hierarchical clustering reveals substantial differences in representation (best network explains about 40% of the variance in human judgments)
When fitting weights that allow the network features to be re-scaled, the network fits much better (best network explains about 84% of the variance in human judgments using out-of-sample predictions)

\[ sim(i, j) = \sum_k w_k f_{ik} f_{jk} \]

\[ w_k : \text{weight for feature } k \]
Predicting neural recordings with a deep convnet


Operations in Linear-Nonlinear Layer

Spatial Convolution over Image Input

1. Optimize Model for Task Performance
2. Test Per-Site Neural Predictions

Behavioral Tasks e.g. Trees vs non-Trees

IT neuronal units vs deep convnet

Similarity matrices for images

Animals (8)
Boats (8)
Cars (8)
Chairs (8)
Faces (8)
Fruits (8)
Planes (8)
Tables (8)

Image generalization

Low Variation Tasks
Medium Variation Tasks
High Variation Tasks

Variation

V1-Like

Model

HMO

LN

IT

V4

V2

100ms Visual Presentation

Neural Recordings from IT and V4

See Commentary
For people, typicality influences performance in practically any category-related task

- speed of categorization
- ease of production
- ease of learning
- usefulness for inductive inference

No known model successfully predicts typicality ratings from raw images -- How do convnets perform?
Category: Banana  ($\rho=0.82$)

How well does this picture fit your idea or image of the category? (rated on 1-7 scale)

**Human typicality ratings**

**Most typical**

- [97.8, 6.8]
- [98.0, 6.8]
- [96.6, 6.8]
- [99.7, 6.6]

- [96.9, 6.6]
- [99.3, 6.0]
- [78.6, 5.8]
- [99.5, 5.5]

- [12.1, 5.3]
- [59.7, 4.4]
- [2.9, 4.3]
- [46.1, 4.1]

- [14.0, 4.1]
- [0.2, 3.6]
- [2.3, 2.5]
- [1.3, 2.4]

**Least typical**

rating key: [machine (0-100), human (1-7)]
Category: Banana ($\rho=0.82$)

How well does this picture fit your idea or image of the category? (rated on 1-7 scale)

**Human typicality ratings**

**Convnet typicality ratings**

rating key: [machine (0-100), human (1-7)]
Category: Bathtub ($\rho=0.68$)

Human typicality ratings

Most typical

[60.6, 6.6]  [58.5, 6.6]  [57.3, 6.6]  [66.5, 6.2]

[72.0, 6.1]  [80.7, 6.0]  [9.5, 5.9]  [35.4, 5.7]

[67.6, 5.6]  [63.0, 5.2]  [9.8, 3.2]  [16.4, 3.1]

[1.0, 3.0]  [1.5, 2.9]  [1.0, 2.8]  [9.1, 2.4]

Least typical

Convnet typicality ratings

[80.7, 6.0]  [72.0, 6.1]  [67.6, 5.6]  [66.5, 6.2]

[63.0, 5.2]  [60.6, 6.6]  [58.5, 6.6]  [57.3, 6.6]

[35.4, 5.7]  [16.4, 3.1]  [9.8, 3.2]  [9.5, 5.9]

[9.1, 2.4]  [1.5, 2.9]  [1.0, 2.8]  [1.0, 3.0]

rating key: [machine (0-100), human (1-7)]
Category: Envelope ($\rho=0.79$)

Human typicality ratings

Most typical

[91.5, 6.7]  [75.2, 6.6]  [96.4, 6.6]  [98.5, 6.6]

[97.7, 6.6]  [82.8, 6.2]  [69.5, 5.3]  [59.7, 5.2]

[31.2, 5.1]  [32.5, 5.1]  [10.8, 4.8]  [5.8, 4.2]

[10.4, 4.1]  [50.6, 3.8]  [41.9, 3.4]  [24.9, 3.2]

Least typical

Convnet typicality ratings

[98.5, 6.6]  [97.7, 6.6]  [96.4, 6.6]  [91.5, 6.7]

[82.8, 6.2]  [75.2, 6.6]  [69.5, 5.3]  [59.7, 5.2]

[50.6, 3.8]  [41.9, 3.4]  [32.5, 5.1]  [31.2, 5.1]

[24.9, 3.2]  [10.8, 4.8]  [10.4, 4.1]  [5.8, 4.2]

rating key: [machine (0-100), human (1-7)]
Category: Teapots ($\rho=0.38$)

Human typicality ratings

Most typical

[95.8, 6.6] [98.8, 6.6] [93.5, 6.4] [98.1, 6.2]

[46.0, 6.0] [63.6, 5.8] [95.0, 5.8] [52.8, 5.8]

[97.2, 5.6] [8.4, 5.3] [93.4, 5.2] [34.9, 4.9]

[78.8, 4.8] [98.5, 4.6] [8.9, 4.3] [83.9, 3.4]

[83.9, 3.4] [78.8, 4.8] [63.6, 5.8] [52.8, 5.8]

[46.0, 6.0] [34.9, 4.9] [8.9, 4.3] [8.4, 5.3]

Least typical

Convnet typicality ratings

[98.8, 6.6] [98.5, 4.6] [98.1, 6.2] [97.2, 5.6]

[95.8, 6.6] [95.0, 5.8] [93.5, 6.4] [93.4, 5.2]

[83.9, 3.4] [78.8, 4.8] [63.6, 5.8] [52.8, 5.8]

[46.0, 6.0] [34.9, 4.9] [8.9, 4.3] [8.4, 5.3]

rating key: [machine (0-100), human (1-7)]
Summary of typicality predictions

<table>
<thead>
<tr>
<th>Item</th>
<th>Rank Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>0.82</td>
</tr>
<tr>
<td>Bathtub</td>
<td>0.68</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>0.62</td>
</tr>
<tr>
<td>Envelope</td>
<td>0.79</td>
</tr>
<tr>
<td>Pillow</td>
<td>0.67</td>
</tr>
<tr>
<td>Soap dispenser</td>
<td>0.74</td>
</tr>
<tr>
<td>Table lamp</td>
<td>0.69</td>
</tr>
<tr>
<td>Teapot</td>
<td>0.38</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.67</strong></td>
</tr>
</tbody>
</table>

Prediction quality varies as a function of network depth.
Critiques of deep convolutional networks

It is easy to fool them with adversarial examples generated to fool networks

(Szegedy et al., 2013)

(Nguyen et al., 2015)
Critiques of deep convolutional networks

Compare to deep convnets, people can learn much richer concepts from less data.

**People learn from less data**
- “one-shot learning”
- where are the others?

**People learn richer concepts**
- parsing
- generating new concepts
  - generating new examples

People learn from less data compared to deep convolutional networks, where people can learn much richer concepts from less data.