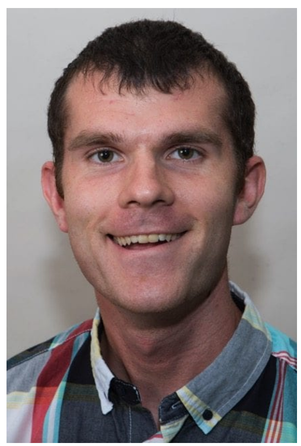
Discovering the Compositional Structure of Vector Representations with Role Learning Networks

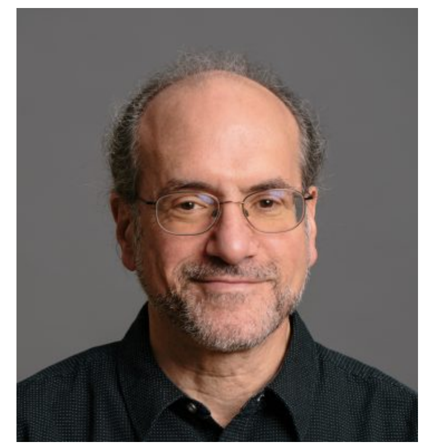
Paul Soulos[1]

Tom McCoy[1]



Tal Linzen[1]

Paul Smolensky[2,1]



1 Johns Hopkins University

2 Microsoft

tl;dr

- Our technique can uncover latent compositionality in vector representations
- Interpreting compositional structure sheds light on how these models function
- We understand the inner workings well enough to write down a symbolic algorithm to produce the neural encoding
- Our approximation allows us to directly manipulate the internal representations to produce desired behavior.

What's in a compositional representation?

Consider a sequence of digits [4, 2, 7, 9]

- A set of fillers (tokens)
  - Example: {4, 2, 7, 9}

What's in a compositional representation?

Consider a sequence of digits [4, 2, 7, 9]

- A set of fillers (tokens)
  - Example: {4, 2, 7, 9}
- A set of roles (positions in the structure)
  - Example: Left-to-right {first, second, third, fourth}

What's in a compositional representation?

Consider a sequence of digits [4, 2, 7, 9]

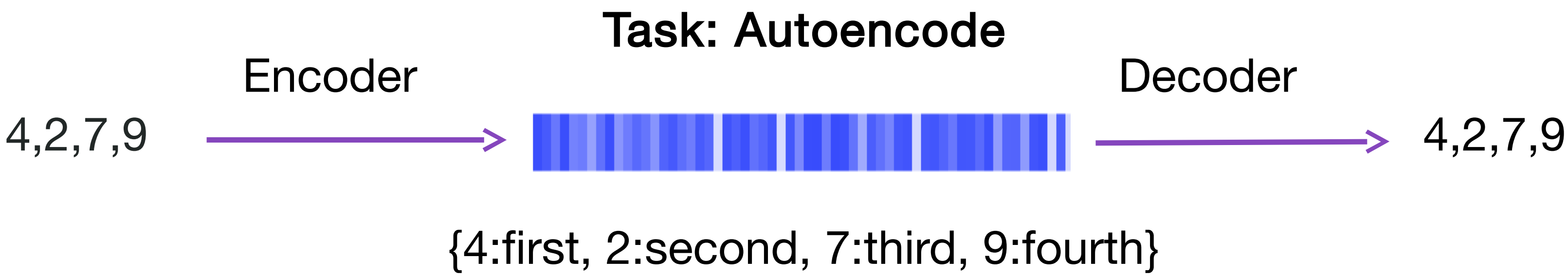
- A set of fillers (tokens)
  - Example: {4, 2, 7, 9}
- A set of roles (positions in the structure)
  - Example: Left-to-right {first, second, third, fourth}
- A binding operation (placing a filler in a specific role filler:role)
  - Example: {4:first}

What's in a compositional representation?

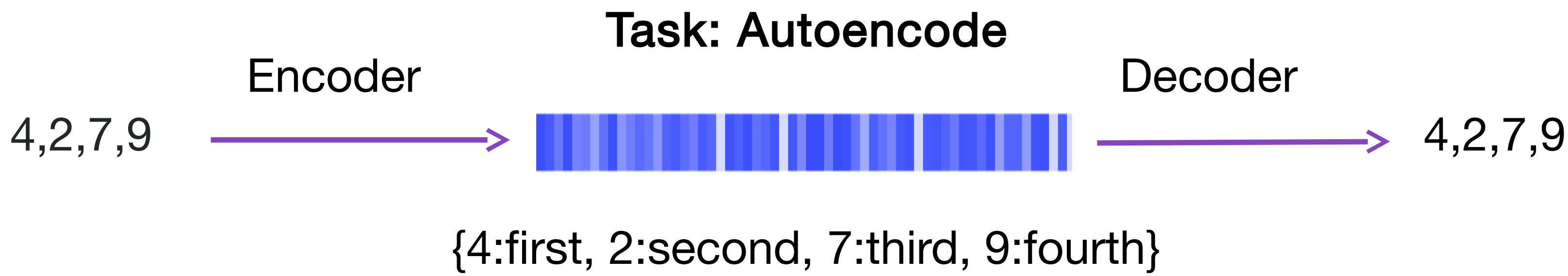
Consider a sequence of digits [4, 2, 7, 9]

- A set of fillers (tokens)
  - Example: {4, 2, 7, 9}
- A set of roles (positions in the structure)
  - Example: Left-to-right {first, second, third, fourth}
- A binding operation (placing a filler in a specific role filler:role)
  - Example: {4:first}
- A composition operation (stitching all of the bound filler:roles together)
  - Example: {4:first, 2:second, 7:third, 9:fourth}

How can neural networks represent compositional structure?



How can neural networks represent compositional structure?



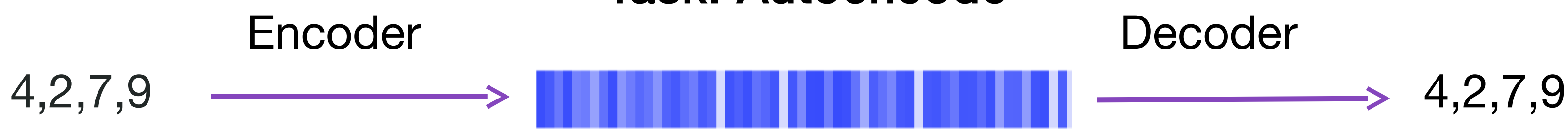
Tensor Product Representations (TPRs)

- A set of fillers (tokens)
- A set of roles (positions in the structure)
- A binding operation (placing a filler in a specific role filler:role)
- A composition operation (stitching all of the bound filler:roles together

Smolensky (1990)

How can neural networks represent compositional structure?

Task: Autoencode



{4:first, 2:second, 7:third, 9:fourth}

Tensor Product Representations (TPRs)

- A set of fillers (tokens)

  Every filler f_i is vector

- A set of roles (positions in the structure)

  Every role r_i is vector

- A binding operation (placing a filler in a specific role filler:role)

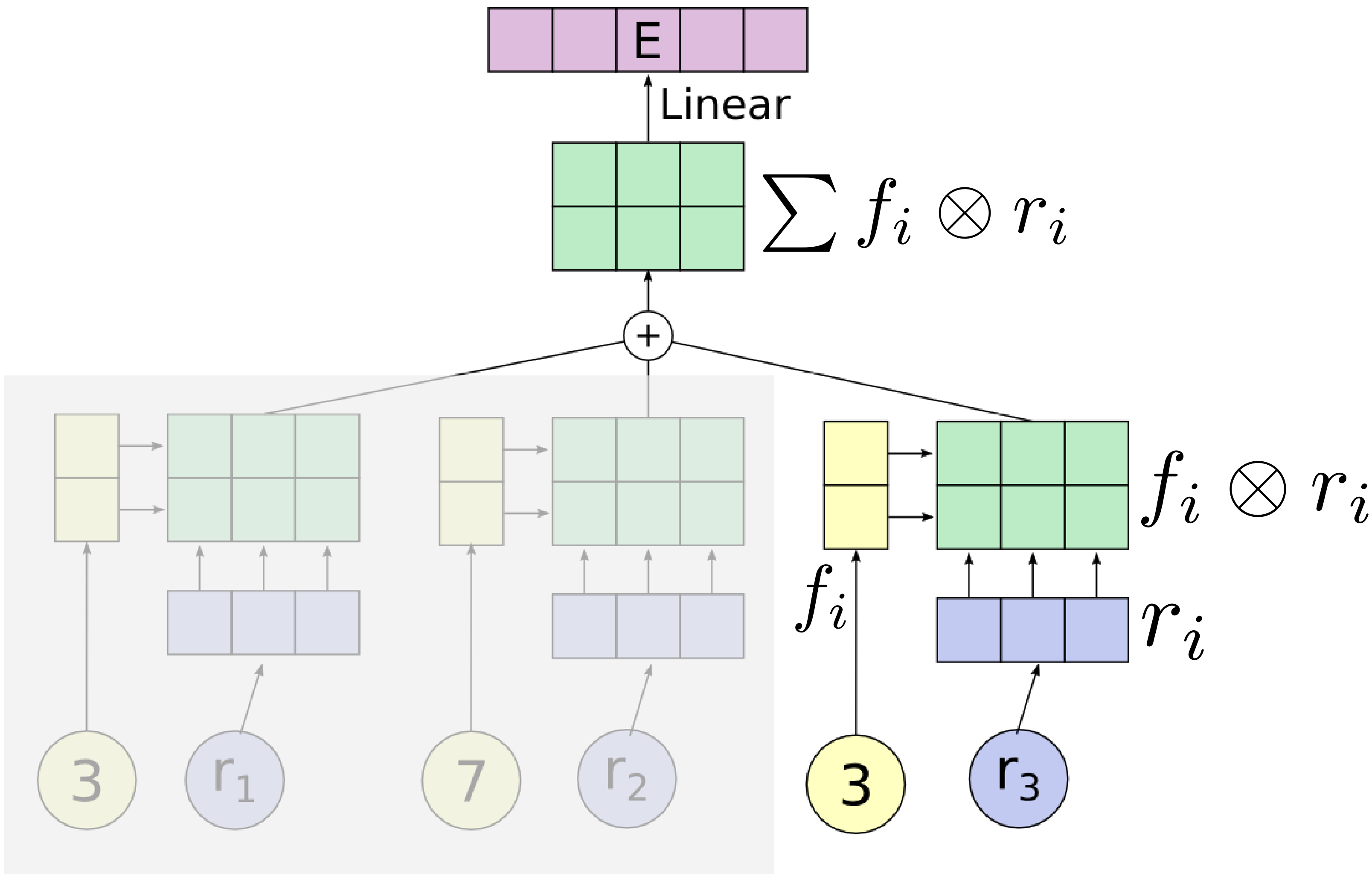
  Tensor product: $f_i \otimes r_i$

- A composition operation (stitching all of the bound filler:roles together

  Sum: $\sum f_i \otimes r_i$

Smolensky (1990)

Tensor Product Encoder

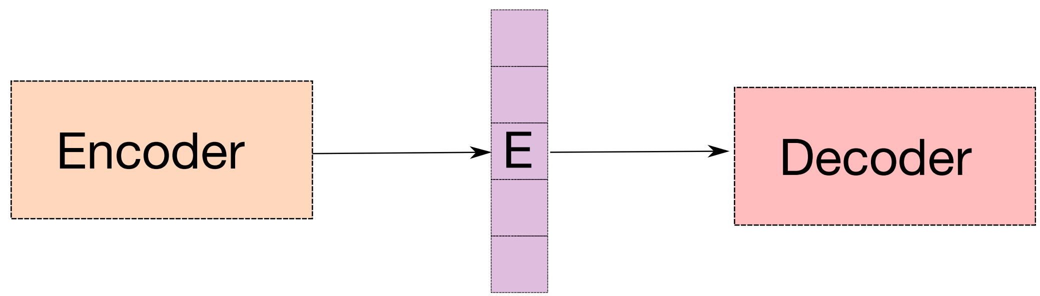


McCoy, Linzen, Dunbar, and Smolensky (2019)

Dissecting Compositionality in Vector Representations (DISCOVER)

Goal: Discover implicit compositional structure in learned encodings E

Target Network

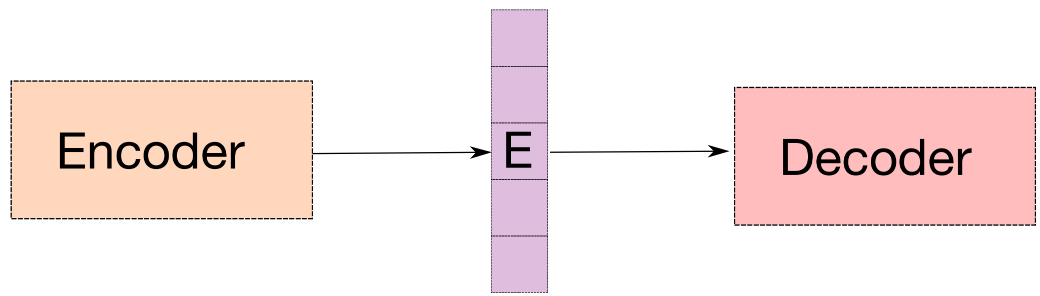


$$E \stackrel{?}{=} \sum f_i \otimes r_i$$

Dissecting Compositionality in Vector Representations (DISCOVER)

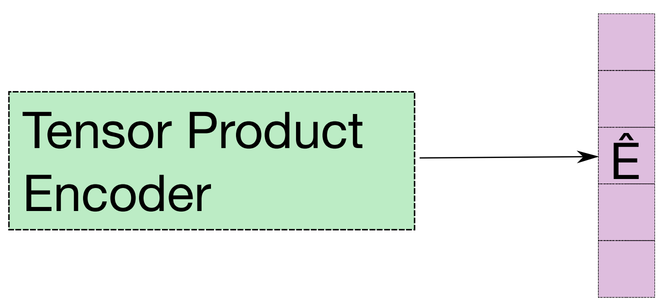
Goal: Discover implicit compositional structure in learned encodings E

Target Network



$$E \stackrel{?}{=} \sum f_i \otimes r_i$$

Approach: Discover implicit compositional structure in the target network's learned encoding E by approximating E with Ê

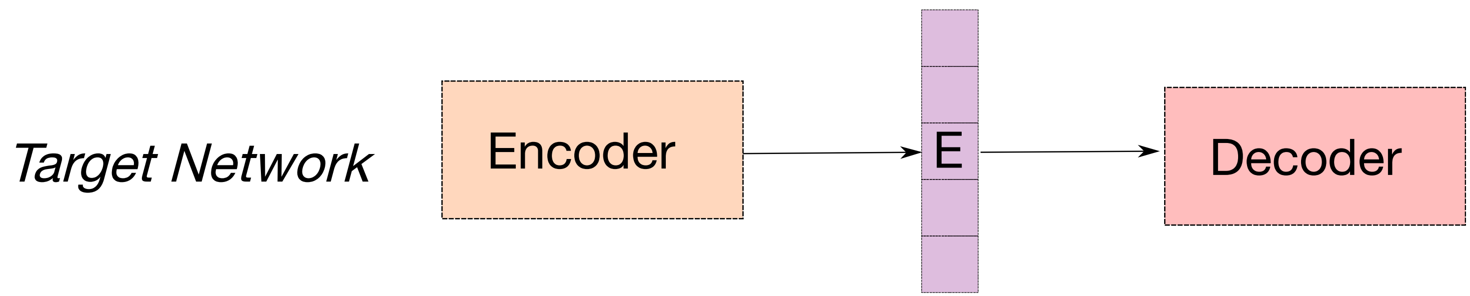


$$\hat{E} = \sum f_i \otimes r_i$$

Minimize MSE(Ê, E)

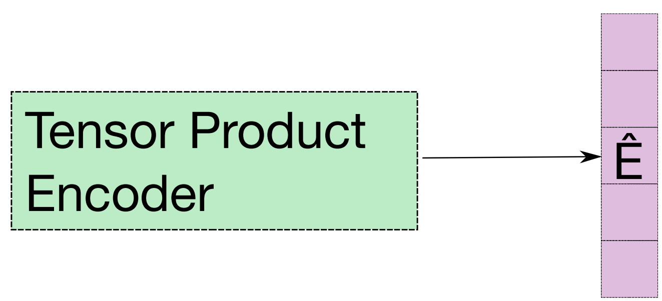
Dissecting Compositionality in Vector Representations (DISCOVER)

Goal: Discover implicit compositional structure in learned encodings E



$$E \stackrel{?}{=} \sum f_i \otimes r_i$$

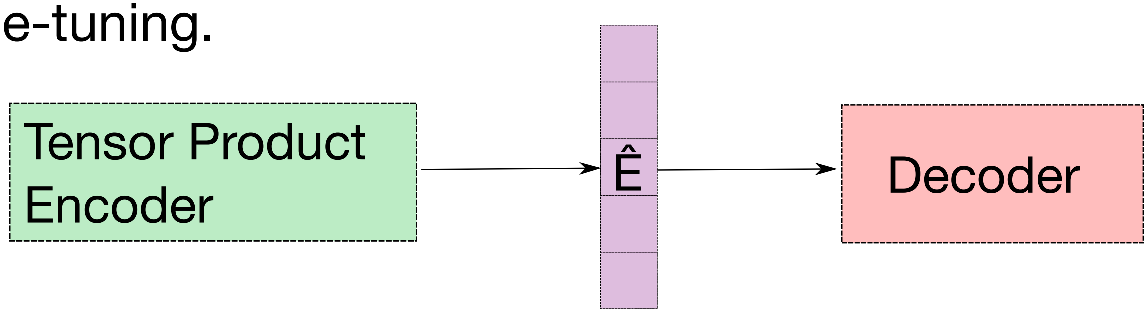
Approach: Discover implicit compositional structure in the target network's learned encoding E by approximating E with Ê



$$\hat{E} = \sum f_i \otimes r_i$$

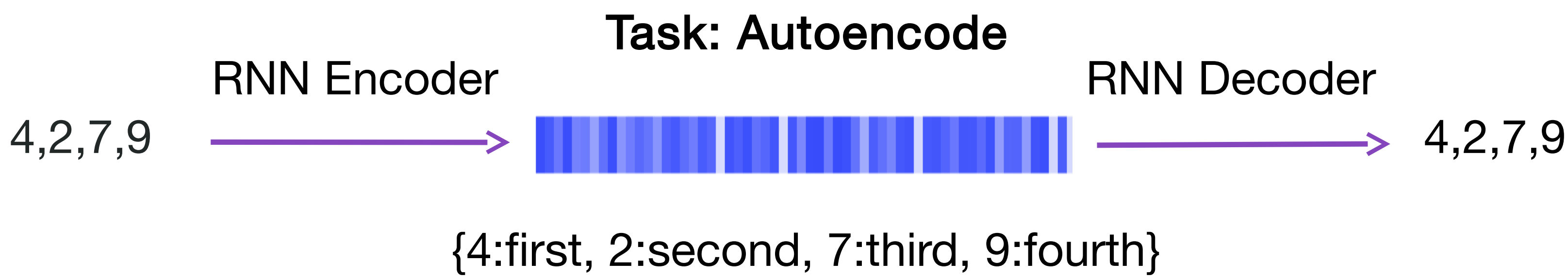
Minimize MSE(Ê, E)

Evaluation: Pass the compositional encoding to the non-compositional decoder. There is no fine-tuning.



$$\hat{E} \stackrel{?}{\approx} E$$

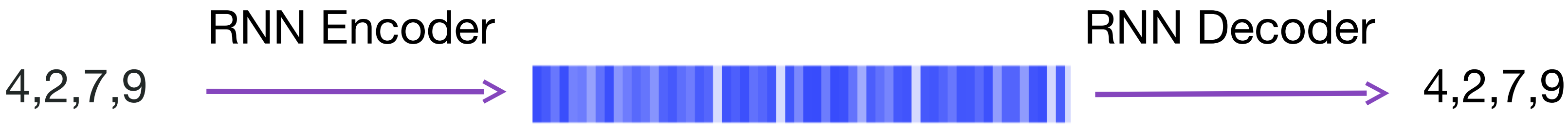
What structure is the target network learning?



Left-to-right (LTR) seems intuitive for copying. We want a FIFO queue to maintain the order.

What structure is the target network learning?

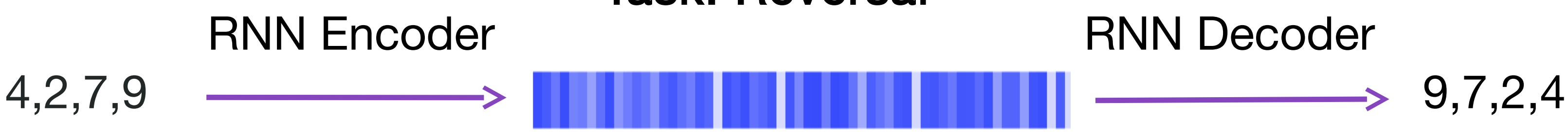
Task: Autoencode



{4:first, 2:second, 7:third, 9:fourth}

Left-to-right (LTR) seems intuitive for copying. We want a FIFO queue to maintain the order.

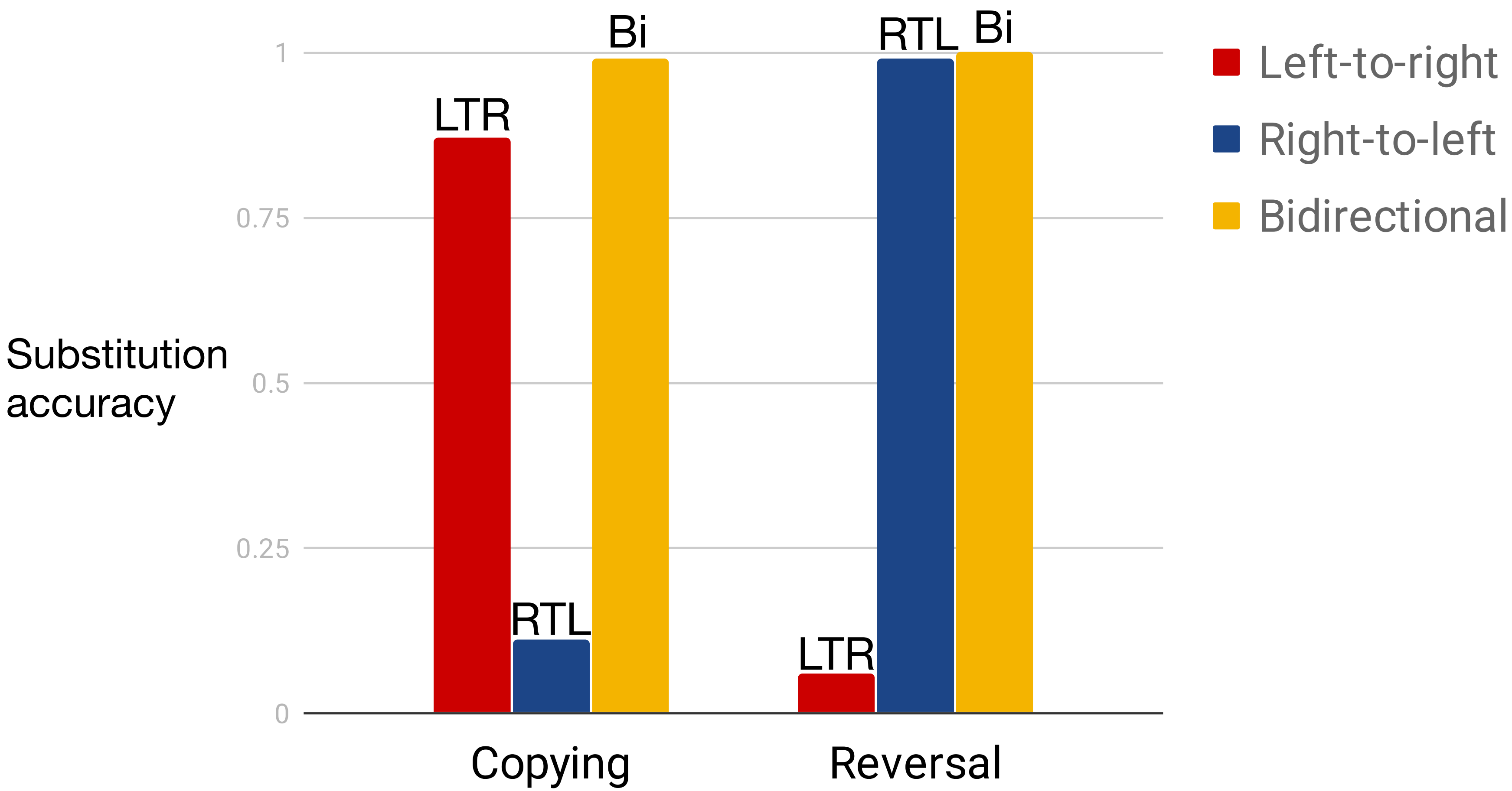
Task: Reversal



{4:fourth, 2:third, 7:second, 9:first}

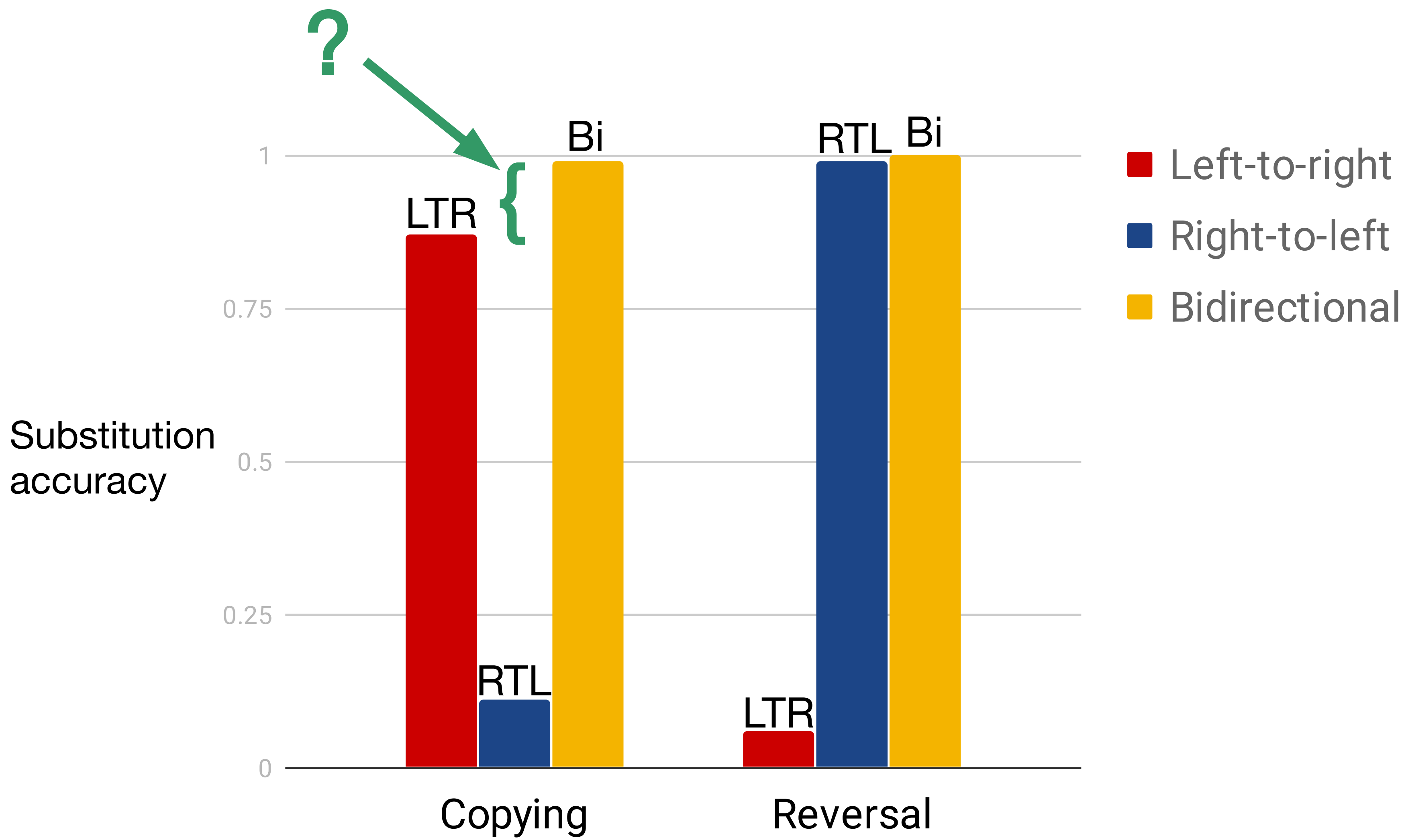
Right-to-left (RTL) seems intuitive for reversal. We want a LIFO stack to reverse the order.

Engineered Roles



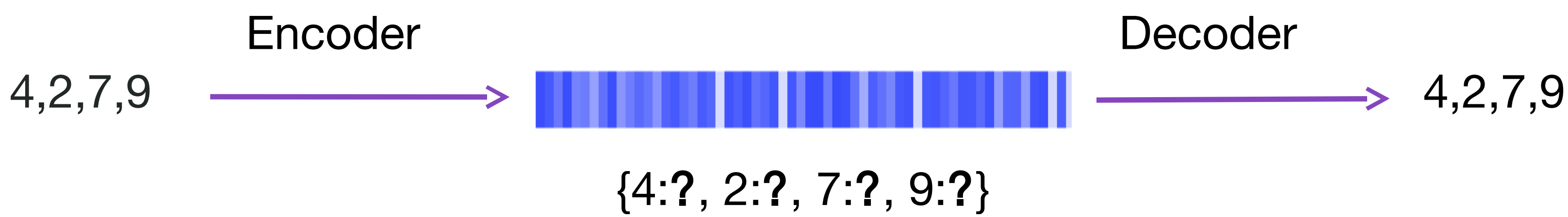
McCoy, Linzen, Dunbar, and Smolensky (2019)

Engineered Roles

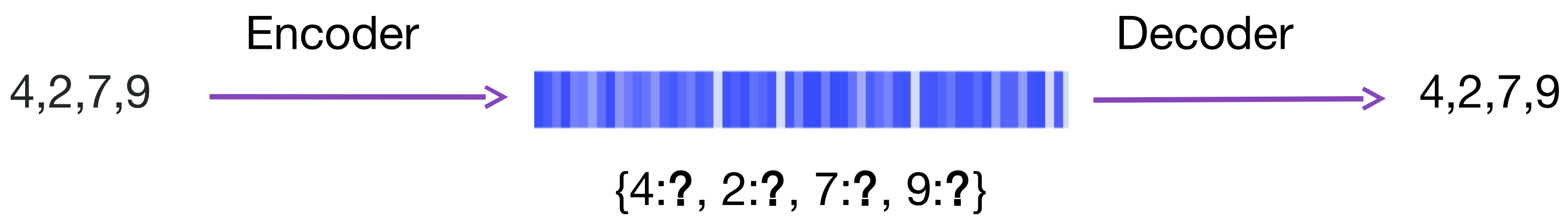


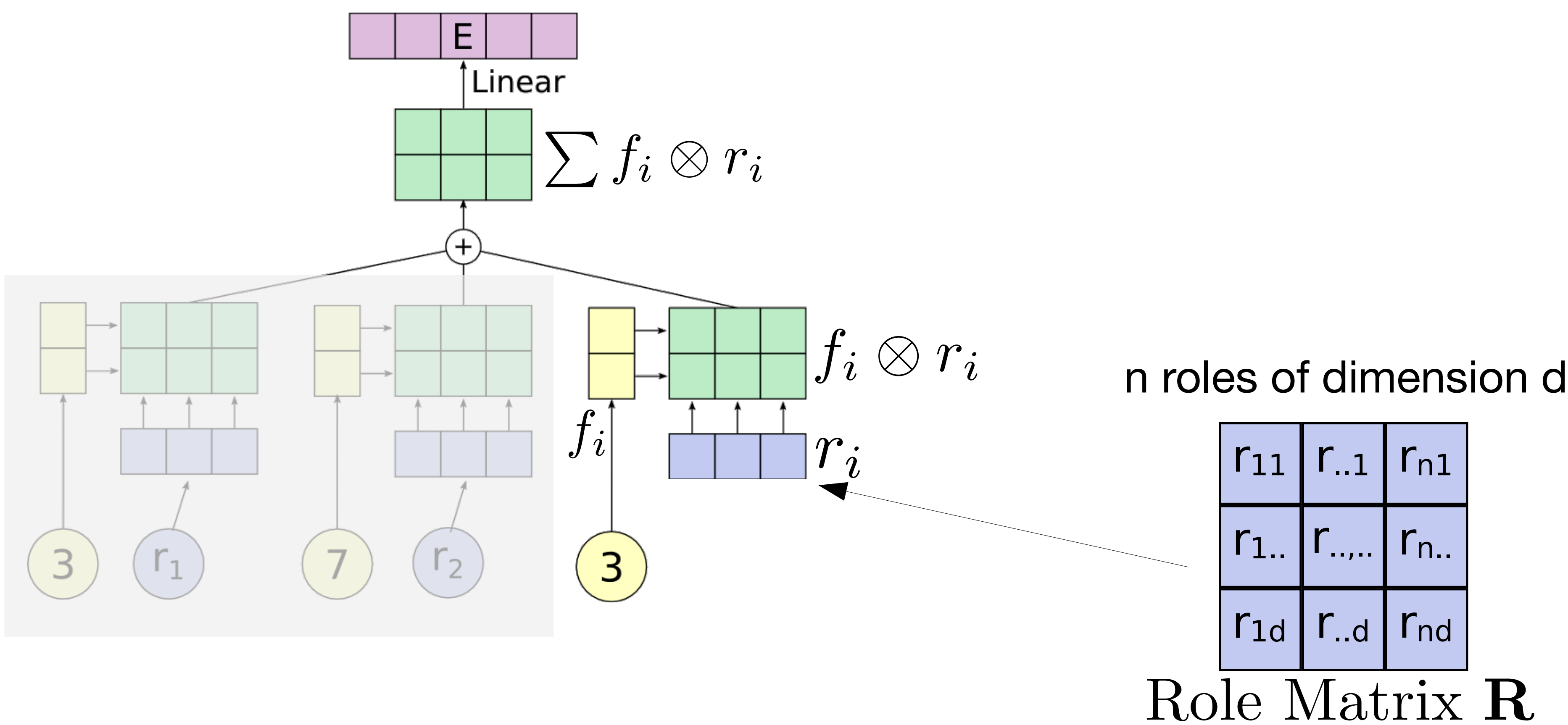
McCoy, Linzen, Dunbar, and Smolensky (2019)

Differentiable Role Assignment

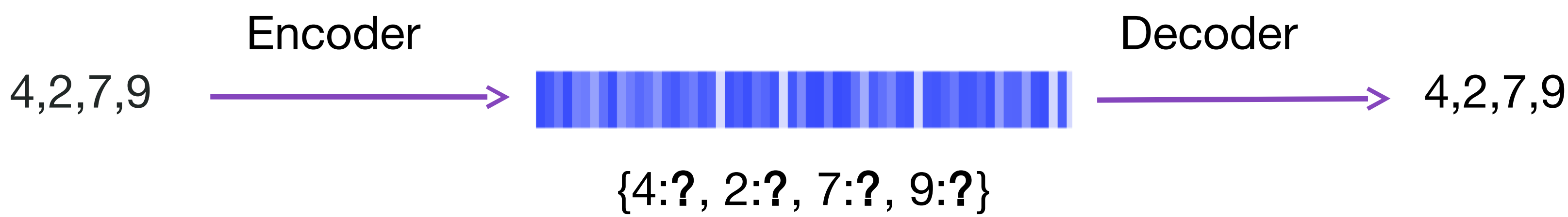


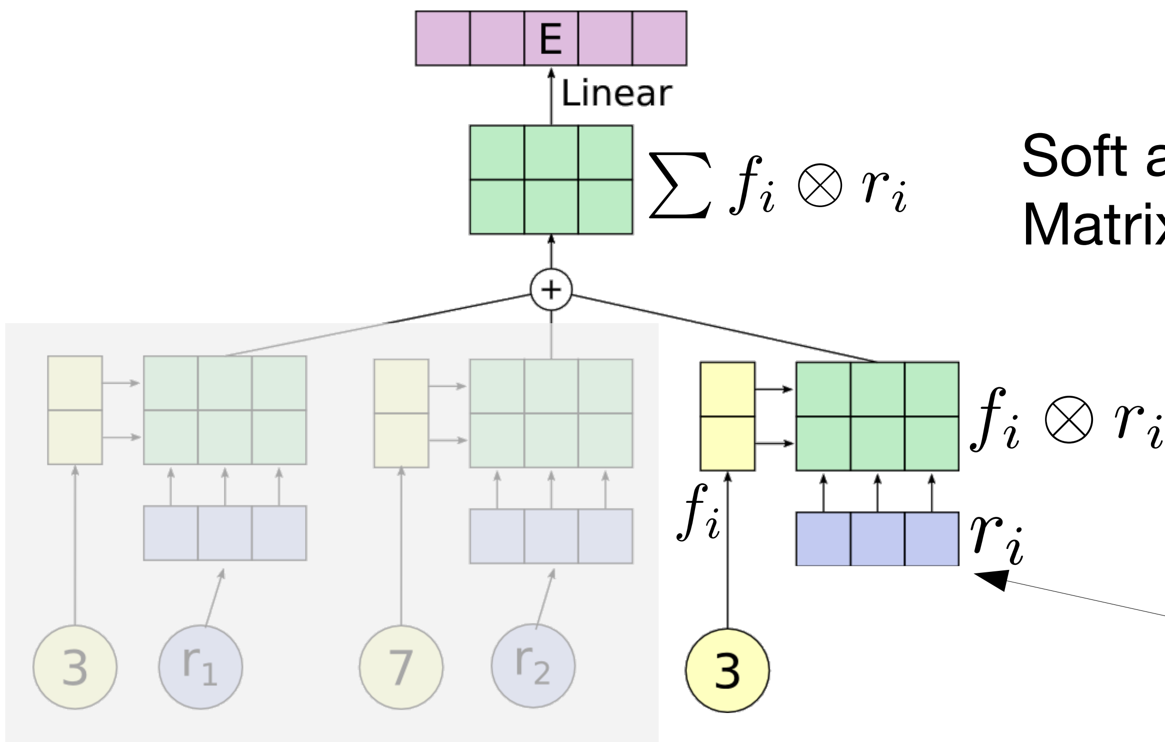
Differentiable Role Assignment



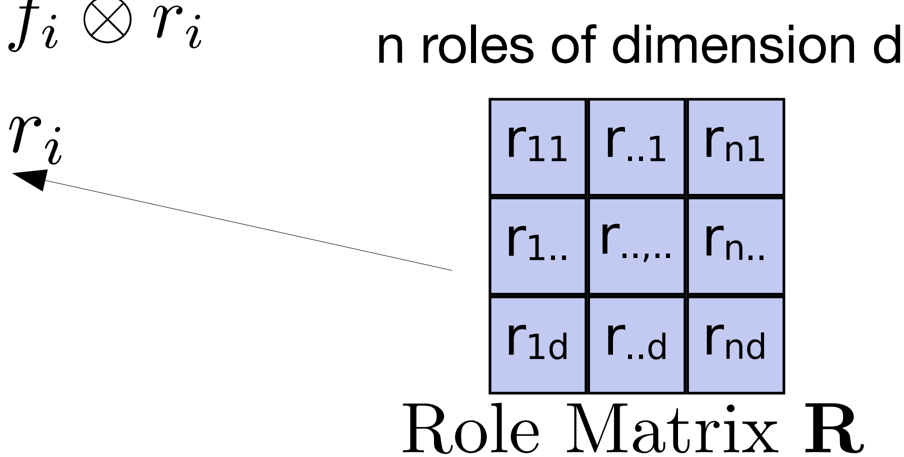


Differentiable Role Assignment





Soft attention over the learned Role Matrix for role assignment



SCAN

Input	Output
jump	JUMP
jump left	LTURN JUMP
jump thrice	JUMP JUMP JUMP
jump opposite left after walk around right	RTURN WALK RTURN WALK RTURN WALK RTURN WALK LTURN LTURN JUMP

Target network is a GRU seq2seq architecture

Lake and Baroni (2018)

SCAN

Input	Output
jump	JUMP
jump left	LTURN JUMP
jump thrice	JUMP JUMP JUMP
jump opposite left after walk around right	RTURN WALK RTURN WALK RTURN WALK RTURN WALK LTURN LTURN JUMP

Target network is a GRU seq2seq architecture

Target	**Learned**	LTR	RTL	Bi	Tree	BOW
98.5%	94.8%	6.7%	7.0%	10.7%	4.3%	4.5%

Table 1: Substitution accuracy for various encoders

Lake and Baroni (2018)

SCAN Role Scheme Interpretation

- Using manual analysis of the role predictions, we created a symbolic algorithm for assigning roles to fillers
- The algorithm matches 98.7% of the role learning network's predictions on the test set.

```python
actions = ["jump", "walk", "look", "run"]
actions_turn = ["jump", "walk", "look", "run", "turn"]
dirs = ["left", "right"]
cards = ["twice", "thrice"]
spins = ["opposite", "around"]
def alg(sequence, correct_roles):
    sequence = sequence.split()
    if "and" in sequence:
        part1 = sequence[:sequence.index("and")]
        part2 = sequence[sequence.index("and") + 1:]
        roles1 = [None for i in range(len(part1))]
        roles1[-1] = '28'
        if len(part1) > 1:
            roles1[0] = '46'
            if part1[1] == "opposite":
                if "thrice" in part1:
                    roles1[1] = '22'
                    roles1[2] = '2'
                else:
                    roles1[1] = '2'
                    roles1[2] = '4'
            elif part1[1] == "around":
                roles1[1] = '22'
                roles1[2] = '2'
            elif part1[1] in dirs:
                if len(part1) == 2:
                    pass#roles1[1] = '28'
                else:
                    roles1[1] = '2'
        if len(part1) == 3:
            roles1[2] = '28'

        roles2 = [None for i in range(len(part2))]
        roles2[0] = '11'
        if len(part2) > 1:
            roles2[-1] = '36'
        if len(part2) > 2:
            roles2[-2] = '3'
        if len(part2) > 3:
            roles2[1] = '24'

        return roles1 + ['30'] + roles2

    elif "after" in sequence:
        part1 = sequence[:sequence.index("after")]
        part2 = sequence[sequence.index("after") + 1:]
        roles1 = [None for i in range(len(part1))]
        roles1[-1] = '8'

        if len(roles1) > 1:
            roles1[-2] = '36'
        if len(roles1) > 2:
            roles1[0] = '11'
        if len(roles1) > 3:
            roles1[1] = '3'

        roles2 = [None for i in range(len(part2))]
        roles2[-1] = '46'

        if len(roles2) > 1:
            roles2[-2] = '4'
        if len(roles2) > 2:
            if part2[-2] == "around":
                if part2[-1] == "right":
                    roles2[0] = '4'
                elif part2[-1] == "left":

                    roles2[0] = '17'
            else:
                if part2[-1] == "thrice" and len(part2) == 4:
                    roles2[0] = '10'
                else:
                    roles2[0] = '17'
        if len(roles2) > 3:
            if part2[-1] == "thrice":
                roles2[1] = '17'
            else:
                roles2[1] = '10'

        if '17' not in roles2 or (part2[-1] == "left" and part2[-2] == "around"):
            return roles1 + ['17'] + roles2
        else:
            return roles1 + ['43'] + roles2
    else:
        roles = [None for i in range(len(sequence))]
        if "twice" in sequence or "thrice" in sequence:
            if len(sequence) == 2 and "twice" in sequence:
                roles[0] = '4'
            else:
                roles[0] = '34'

            if len(sequence) == 2 and "thrice" in sequence:
                roles[-1] = '2'
            else:
                roles[-1] = '46'
            if len(sequence) > 1:
                if sequence[1] in spins:
                    if sequence[1] == "opposite" and "twice" in sequence:
                        roles[1] = '8'
                    elif sequence[1] == "opposite" and "thrice" in sequence:
                        roles[1] = '34'
                    else:
                        roles[1] = '22'
                if sequence[-2] in dirs:
                    roles[-2] = '2'

        else:
            roles[0] = '46'
            if len(sequence) > 1:
                if sequence[1] == "opposite":
                    roles[1] = '2'
                    roles[2] = '26'
                elif sequence[1] == "around":
                    roles[1] = '3'
                    roles[2] = '22'
                if sequence[1] in dirs:
                    roles[1] = '22'

        return roles
```

SCAN Role Scheme Interpretation

- Using manual analysis of the role predictions, we created a symbolic algorithm for assigning roles to fillers
- The algorithm matches 98.7% of the role learning network's predictions on the test set.
- Most roles are defined based on position in a subclause (e.g. *last element of the first subclause*)
- **Example roles:**
  - Role 30: Always assigned to *and*
  - Role 17: Only appears in sequences that contain the word *after*
- These two roles allow the decoder to understand the basic syntax of the command.

```python
actions = ["jump", "walk", "look", "run"]
actions_turn = ["jump", "walk", "look", "run", "turn"]
dirs = ["left", "right"]
cards = ["twice", "thrice"]
spins = ["opposite", "around"]
def alg(sequence, correct_roles):
    sequence = sequence.split()
    if "and" in sequence:
        part1 = sequence[:sequence.index("and")]
        part2 = sequence[sequence.index("and") + 1:]
        roles1 = [None for i in range(len(part1))]
        roles1[-1] = '28'
        if len(part1) > 1:
            roles1[0] = '46'
            if part1[1] == "opposite":
                if "thrice" in part1:
                    roles1[1] = '22'
                    roles1[2] = '2'
                else:
                    roles1[1] = '2'
                    roles1[2] = '4'
            elif part1[1] == "around":
                roles1[1] = '22'
                roles1[2] = '2'
            elif part1[1] in dirs:
                if len(part1) == 2:
                    pass#roles1[1] = '28'
                else:
                    roles1[1] = '2'
        if len(part1) == 3:
            roles1[2] = '28'

        roles2 = [None for i in range(len(part2))]
        roles2[0] = '11'
        if len(part2) > 1:
            roles2[-1] = '36'
        if len(part2) > 2:
            roles2[-2] = '3'
        if len(part2) > 3:
            roles2[1] = '24'

        return roles1 + ['30'] + roles2

    elif "after" in sequence:
        part1 = sequence[:sequence.index("after")]
        part2 = sequence[sequence.index("after") + 1:]
        roles1 = [None for i in range(len(part1))]
        roles1[-1] = '8'

        if len(roles1) > 1:
            roles1[-2] = '36'
        if len(roles1) > 2:
            roles1[0] = '11'
        if len(roles1) > 3:
            roles1[1] = '3'

        roles2 = [None for i in range(len(part2))]
        roles2[-1] = '46'

        if len(roles2) > 1:
            roles2[-2] = '4'
        if len(roles2) > 2:
            if part2[-2] == "around":
                if part2[-1] == "right":
                    roles2[0] = '4'
                elif part2[-1] == "left":

                    roles2[0] = '17'
            else:
                if part2[-1] == "thrice" and len(part2) == 4:
                    roles2[0] = '10'
                else:
                    roles2[0] = '17'
        if len(roles2) > 3:
            if part2[-1] == "thrice":
                roles2[1] = '17'
            else:
                roles2[1] = '10'

        if '17' not in roles2 or (part2[-1] == "left" and part2[-2] == "around"):
            return roles1 + ['17'] + roles2
        else:
            return roles1 + ['43'] + roles2
    else:
        roles = [None for i in range(len(sequence))]
        if "twice" in sequence or "thrice" in sequence:
            if len(sequence) == 2 and "twice" in sequence:
                roles[0] = '4'
            else:
                roles[0] = '34'

            if len(sequence) == 2 and "thrice" in sequence:
                roles[-1] = '2'
            else:
                roles[-1] = '46'
            if len(sequence) > 1:
                if sequence[1] in spins:
                    if sequence[1] == "opposite" and "twice" in sequence:
                        roles[1] = '8'
                    elif sequence[1] == "opposite" and "thrice" in sequence:
                        roles[1] = '34'
                    else:
                        roles[1] = '22'
                if sequence[-2] in dirs:
                    roles[-2] = '2'

        else:
            roles[0] = '46'
            if len(sequence) > 1:
                if sequence[1] == "opposite":
                    roles[1] = '2'
                    roles[2] = '26'
                elif sequence[1] == "around":
                    roles[1] = '3'
                    roles[2] = '22'
                if sequence[1] in dirs:
                    roles[1] = '22'

        return roles
```

Differentiable API Design

- Consider SCAN as a coding assignment between a pair of students.
  - Let's call them “Encoder” and “Decoder”

Differentiable API Design

- Consider SCAN as a coding assignment between a pair of students.
  - Let's call them “Encoder” and “Decoder”
- They split the assignment so that Encoder parses the input into a data structure, and Decoder produces the output from this data structure

```
? encode(List<Input Tokens>)

List<Output Tokens> decode(?)
```

Differentiable API Design

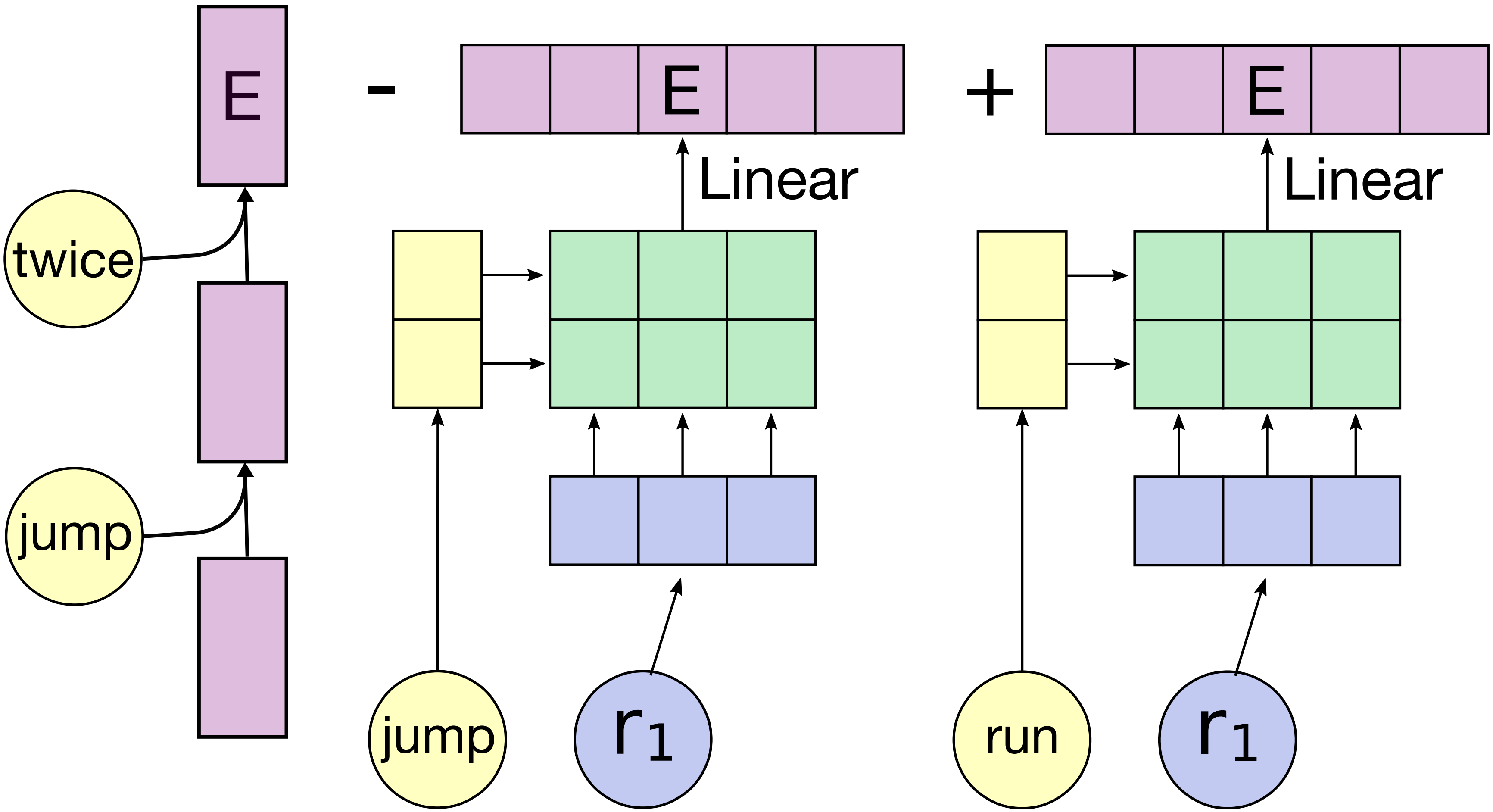
- Consider SCAN as a coding assignment between a pair of students.
  - Let's call them “Encoder” and “Decoder”
- They split the assignment so that Encoder parses the input into a data structure, and Decoder produces the output from this data structure

```
? encode(List<Input Tokens>)

List<Output Tokens> decode(?)
```

```
<isAnd, isAfter, subclauseOneAction, subclauseOneSecondWord...> encode(List<Input Tokens>)

List<Output Tokens> decode(<isAnd, isAfter, subclauseOneAction, subclauseOneSecondWord...>)
```

Constituent Surgery

$$\text{emb(jump twice)} - \text{TPR(jump)} + \text{TPR(run)} \stackrel{?}{=} \text{emb(run twice)}$$



JUMP JUMP → RUN RUN

Constituent Surgery

$\texttt{run}:11$ $\texttt{left}:36$ $\texttt{twice}:8$ $\texttt{after}:43$ $\texttt{jump}:10$ $\texttt{opposite}:17$ $\texttt{right}:4$ $\texttt{thrice}:46 \rightarrow$
TR TR JUMP TR TR JUMP TR TR JUMP TL RUN TL RUN
$-\ \texttt{run}:11\ +\ \texttt{look}:11 \rightarrow$
TR TR JUMP TR TR JUMP TR TR JUMP TL LOOK TL LOOK

Constituent Surgery

$\texttt{run}:11\ \texttt{left}:36\ \texttt{twice}:8\ \texttt{after}:43\ \texttt{jump}:10\ \texttt{opposite}:17\ \texttt{right}:4\ \texttt{thrice}:46 \rightarrow$

TR TR JUMP TR TR JUMP TR TR JUMP TL RUN TL RUN

$-\,\texttt{run}:11\ \ +\texttt{look}:11 \rightarrow$

TR TR JUMP TR TR JUMP TR TR JUMP TL LOOK TL LOOK

$-\,\texttt{jump}:10\ \ +\texttt{walk}:10 \rightarrow$

TR TR WALK TR TR WALK TR TR WALK TL LOOK TL LOOK

Constituent Surgery

$\texttt{run}:11\ \texttt{left}:36\ \texttt{twice}:8\ \texttt{after}:43\ \texttt{jump}:10\ \texttt{opposite}:17\ \texttt{right}:4\ \texttt{thrice}:46 \rightarrow$

TR TR JUMP TR TR JUMP TR TR JUMP TL RUN TL RUN

$-\ \texttt{run}:11\ \ +\texttt{look}:11 \rightarrow$

TR TR JUMP TR TR JUMP TR TR JUMP TL LOOK TL LOOK

$-\ \texttt{jump}:10\ \ +\texttt{walk}:10 \rightarrow$

TR TR WALK TR TR WALK TR TR WALK TL LOOK TL LOOK

$-\ \texttt{left}:36\ \ +\texttt{right}:36 \rightarrow$

TR TR WALK TR TR WALK TR TR WALK TR LOOK TR LOOK

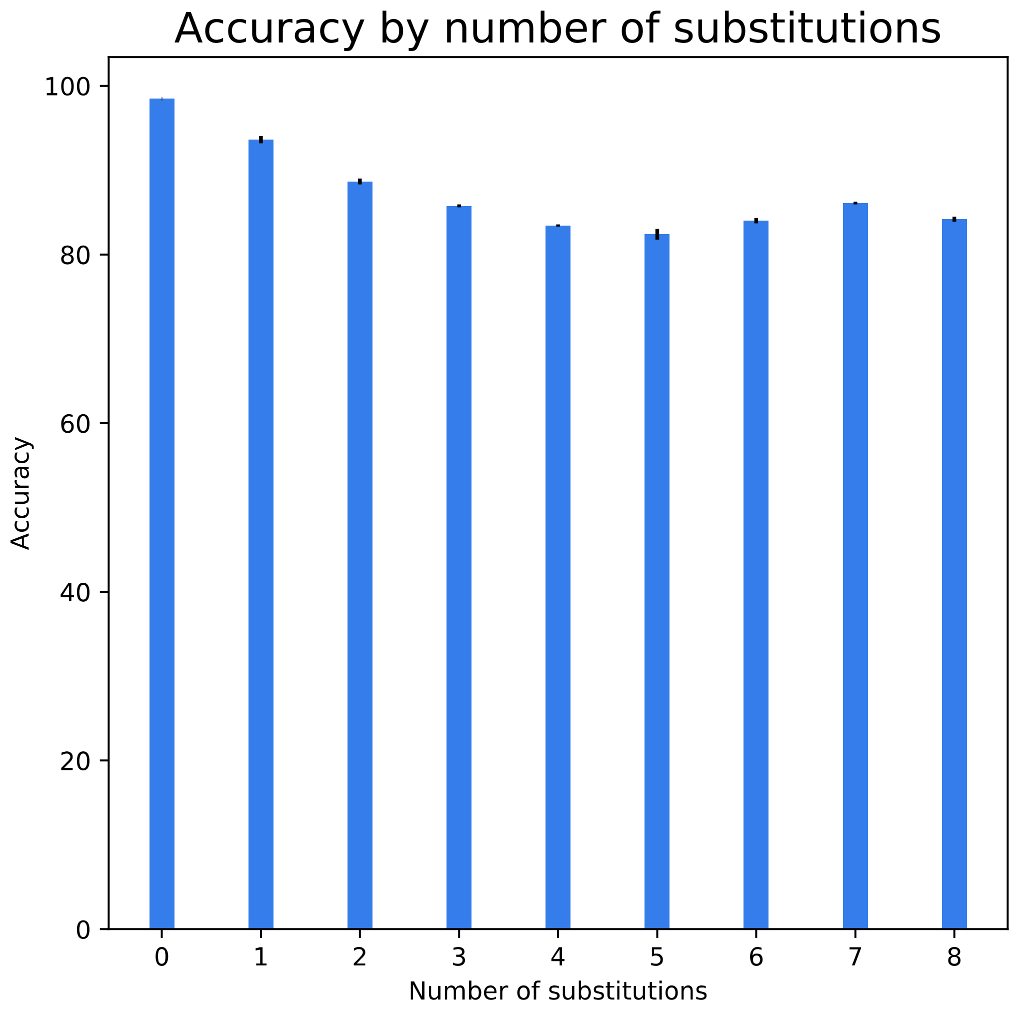
Constituent Surgery

$\texttt{run}:11\ \texttt{left}:36\ \texttt{twice}:8\ \texttt{after}:43\ \texttt{jump}:10\ \texttt{opposite}:17\ \texttt{right}:4\ \texttt{thrice}:46 \rightarrow$
TR TR JUMP TR TR JUMP TR TR JUMP TL RUN TL RUN
$-\ \texttt{run}:11\ +\texttt{look}:11 \rightarrow$
TR TR JUMP TR TR JUMP TR TR JUMP TL LOOK TL LOOK
$-\ \texttt{jump}:10\ +\texttt{walk}:10 \rightarrow$
TR TR WALK TR TR WALK TR TR WALK TL LOOK TL LOOK
$-\ \texttt{left}:36\ +\texttt{right}:36 \rightarrow$
TR TR WALK TR TR WALK TR TR WALK TR LOOK TR LOOK
$-\ \texttt{twice}:8\ +\texttt{thrice}:8 \rightarrow$
TR TR WALK TR TR WALK TR TR WALK TR LOOK TR LOOK TR LOOK

Constituent Surgery

run : 11 left : 36 twice : 8 after : 43 jump : 10 opposite : 17 right : 4 thrice : 46 $\rightarrow$
TR TR JUMP TR TR JUMP TR TR JUMP TL RUN TL RUN
$-$ run : 11 $+$ look : 11 $\rightarrow$
TR TR JUMP TR TR JUMP TR TR JUMP TL LOOK TL LOOK
$-$ jump : 10 $+$ walk : 10 $\rightarrow$
TR TR WALK TR TR WALK TR TR WALK TL LOOK TL LOOK
$-$ left : 36 $+$ right : 36 $\rightarrow$
TR TR WALK TR TR WALK TR TR WALK TR LOOK TR LOOK
$-$ twice : 8 $+$ thrice : 8 $\rightarrow$
TR TR WALK TR TR WALK TR TR WALK TR LOOK TR LOOK TR LOOK
$-$ opposite : 17 $+$ around : 17 $\rightarrow$
TR WALK TR WALK TR WALK TR WALK TR WALK TR WALK TR WALK TR WALK TR WALK TR WALK
TR WALK TR WALK TR LOOK TR LOOK TR LOOK

Constituent Surgery (Continued)



Sentence Embedding Models

	Learned	LTR	RTL	Bi	Tree	BOW
InferSent	**4.05e-4**	8.21e-4	9.70e-4	9.16e-4	7.78e-4	4.34e-4
Skip-thought	9.30e-5	9.91e-5	1.78e-3	3.95e-4	9.64e-5	**8.87e-5**
SST	**5.58e-3**	8.35e-3	9.29e-3	8.55e-3	5.99e-3	9.38e-3
SPINN	**.139**	.184	.189	.181	.178	.176

Mean-squared error for learned and engineered role schemes.

Future Directions

- Train the Tensor Product Encoder end-to-end
- Tensor Product Decoder
- Does a compositional bias improve training?
  - Train faster, fewer parameters, better generalization
- Improving natural language models with a compositional bias

Thank you!

- Run the code yourself
  - https://github.com/psoulos/role-decomposition

- Want more details?
  - Come by the poster
  - Check out the paper: https://arxiv.org/abs/1910.09113

Acknowledgements

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 1746891, and work partially supported by NSF INSPIRE grant BCS-1344269. All opinions are our own.