ITCS 6156/8156 Spring 2024
Machine Learning

Recurrent Neural Networks

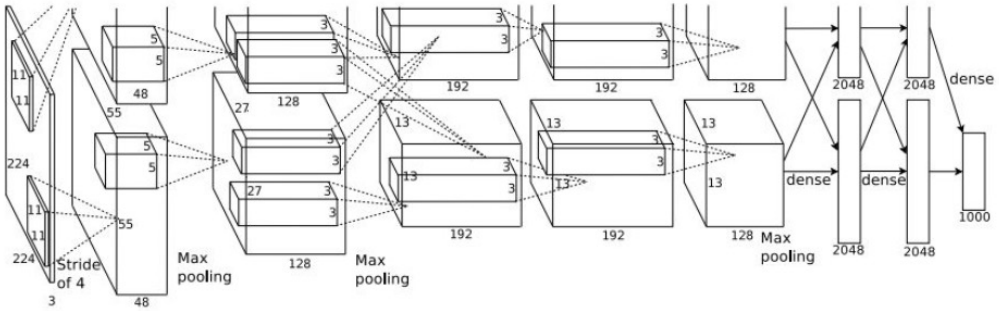
Instructor: Hongfei Xue
Email: hongfei.xue@charlotte.edu
Class Meeting: Mon & Wed, 4:00 PM – 5:15 PM, Denny 109

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

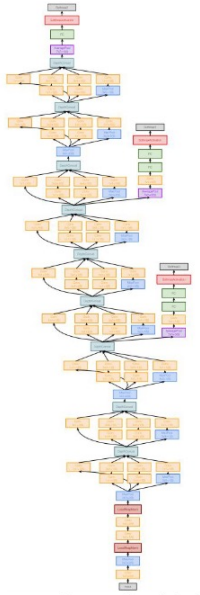
Some content in the slides is based on DeepMind's and Dr. Fei-Fei Li's lectures

Recap: CNN Architecture

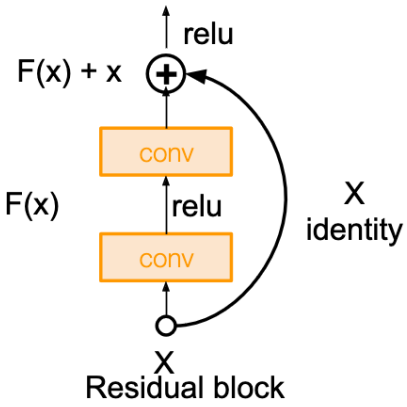
AlexNet

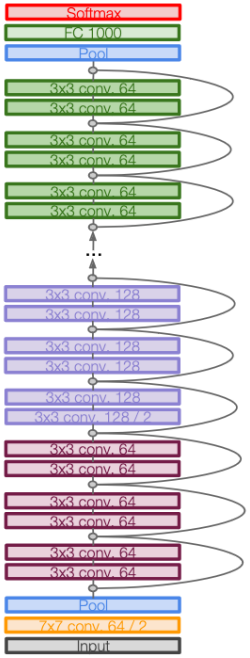


VGG16 VGG19

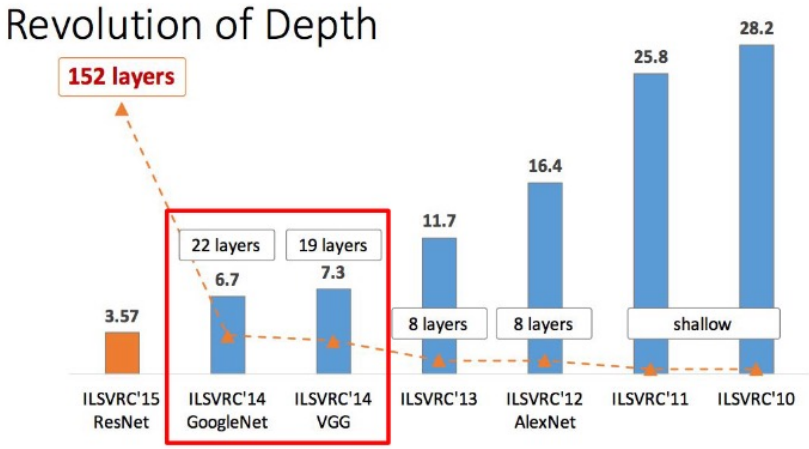


GoogLeNet

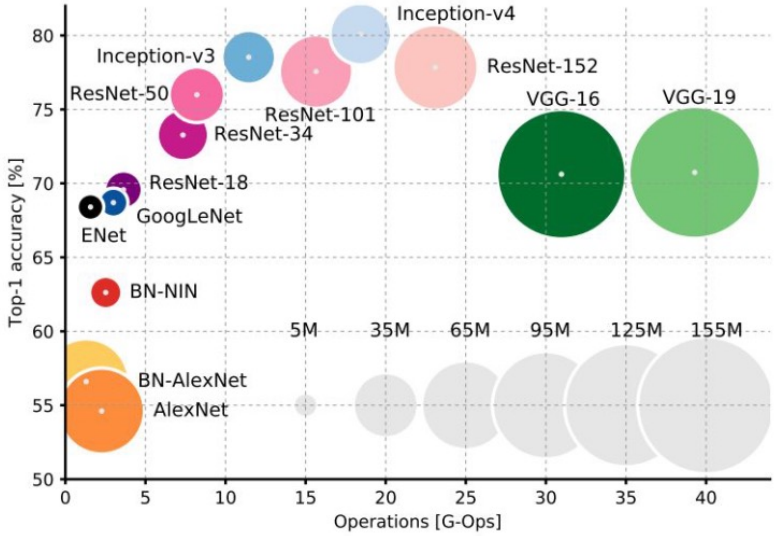




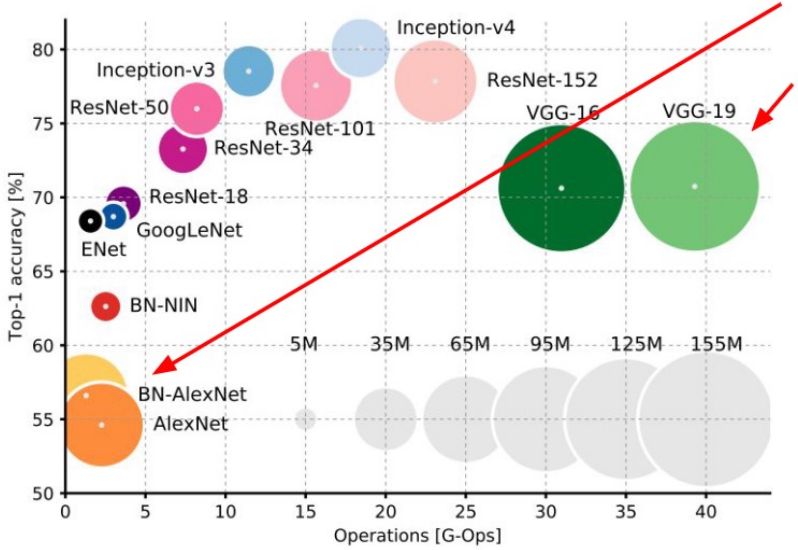
ResNet



Recap: CNN Architecture



Recap: CNN Architecture

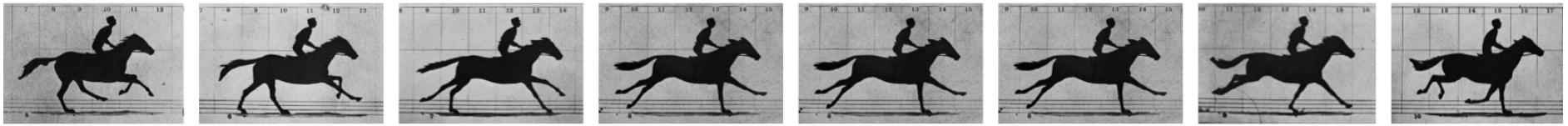


AlexNet and VGG have tons of parameters in the fully connected layers

AlexNet: ~62M parameters

FC6: 256x6x6 -> 4096: 38M params
FC7: 4096 -> 4096: 17M params
FC8: 4096 -> 1000: 4M params
~59M params in FC layers!

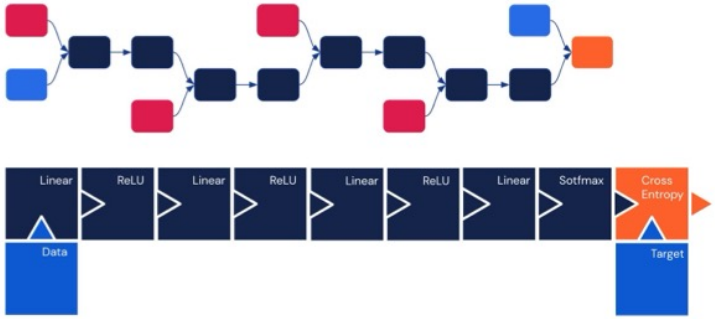
Modeling Sequences

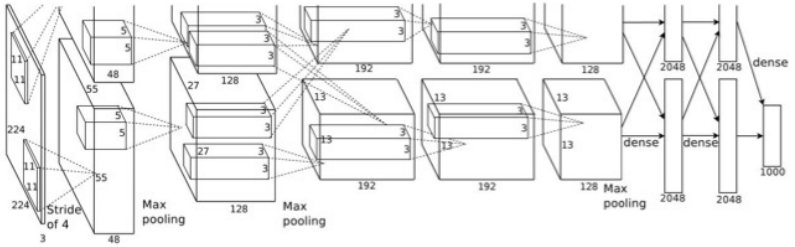


Collections of elements where:

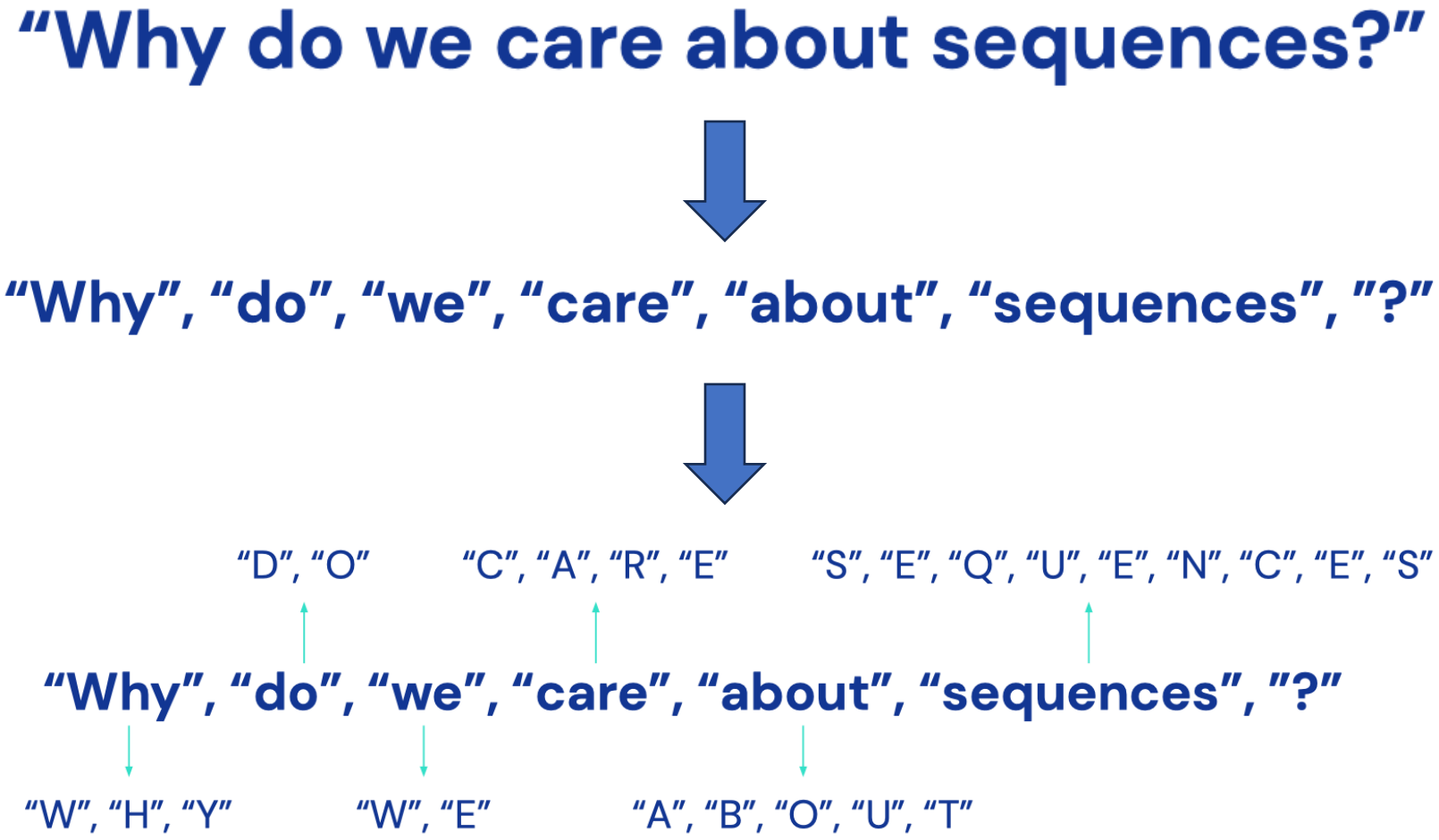
- Elements can be **repeated**
- **Order** matters
- Of **variable** (potentially infinite) length

Models discussed so far don't do well with sequential data.

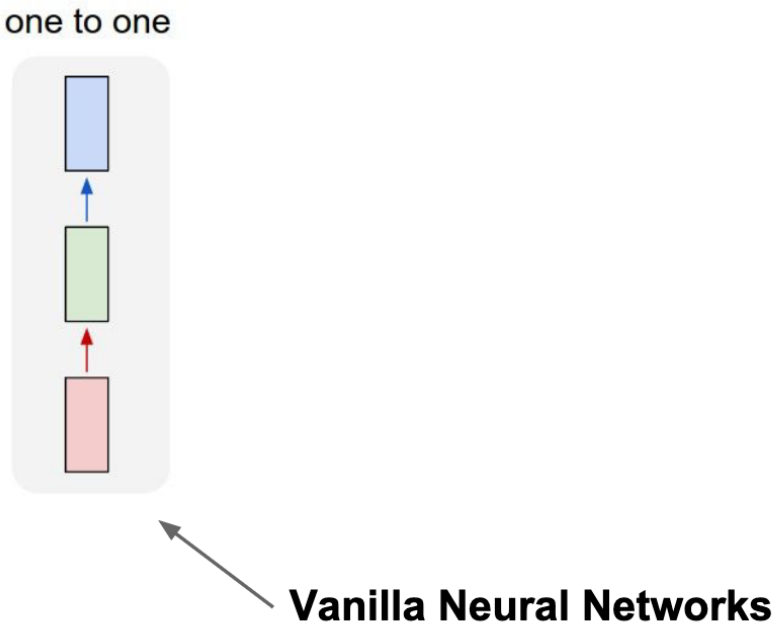




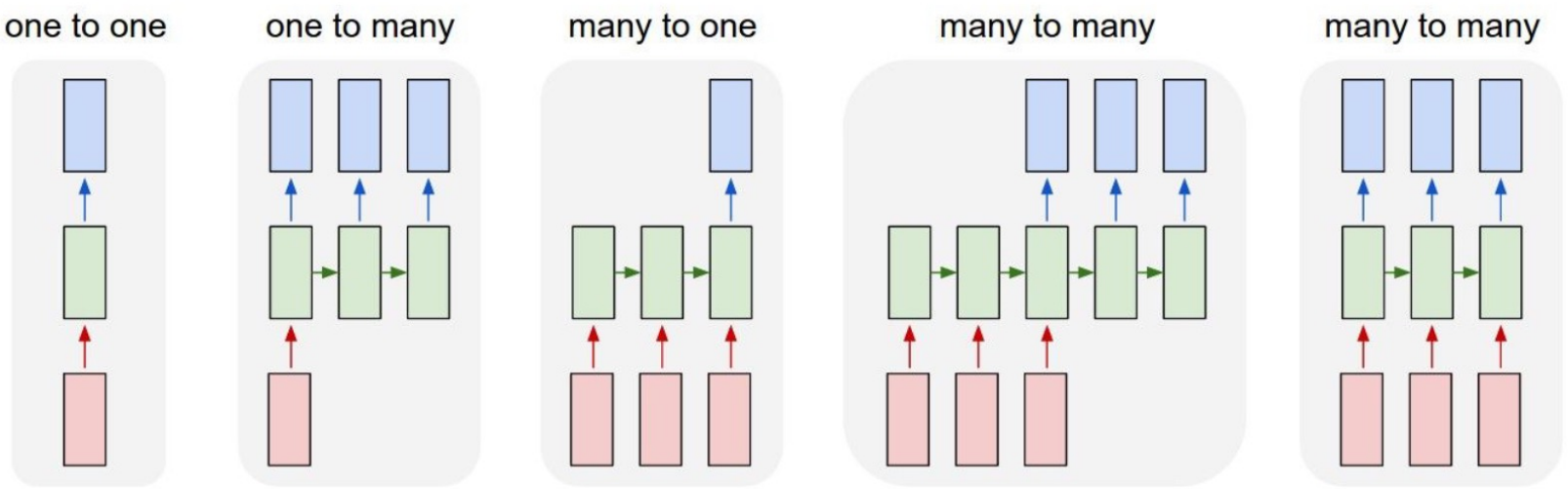
Modeling Sequences



Process Sequence

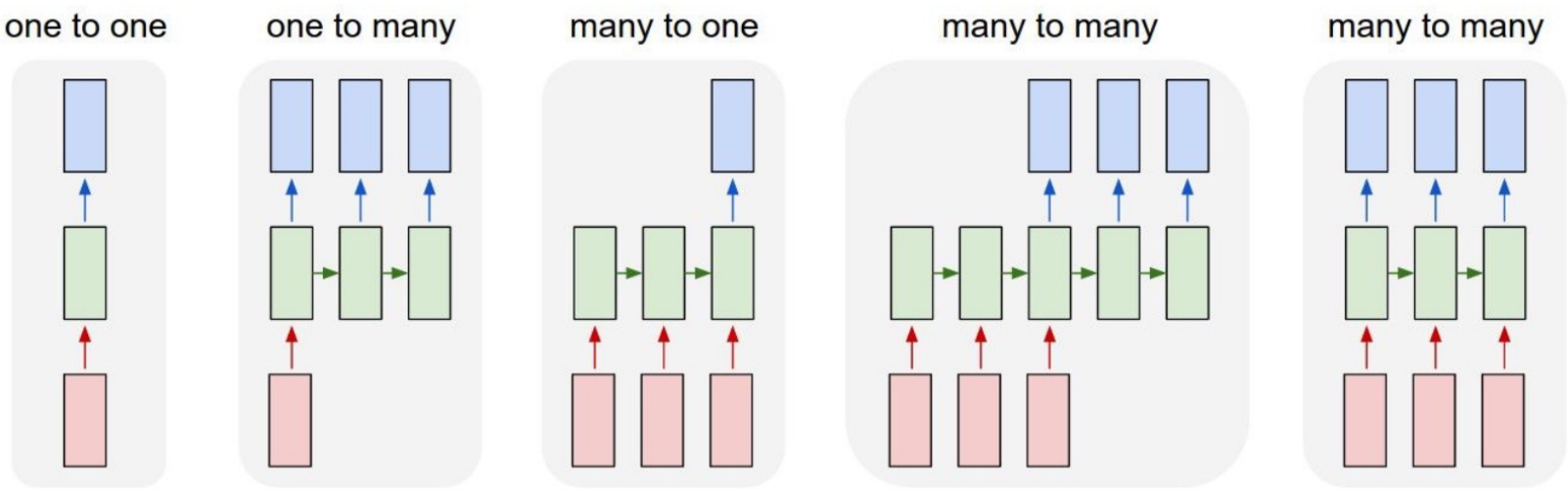


Process Sequence



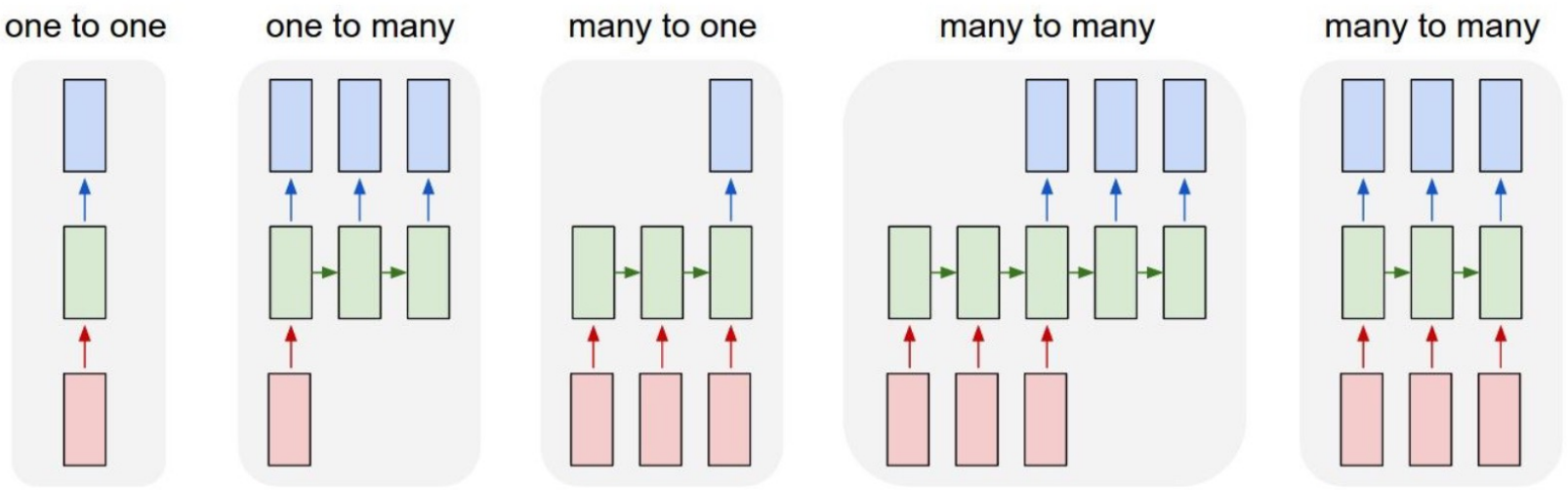
e.g. **Image Captioning**
image -> sequence of words

Process Sequence



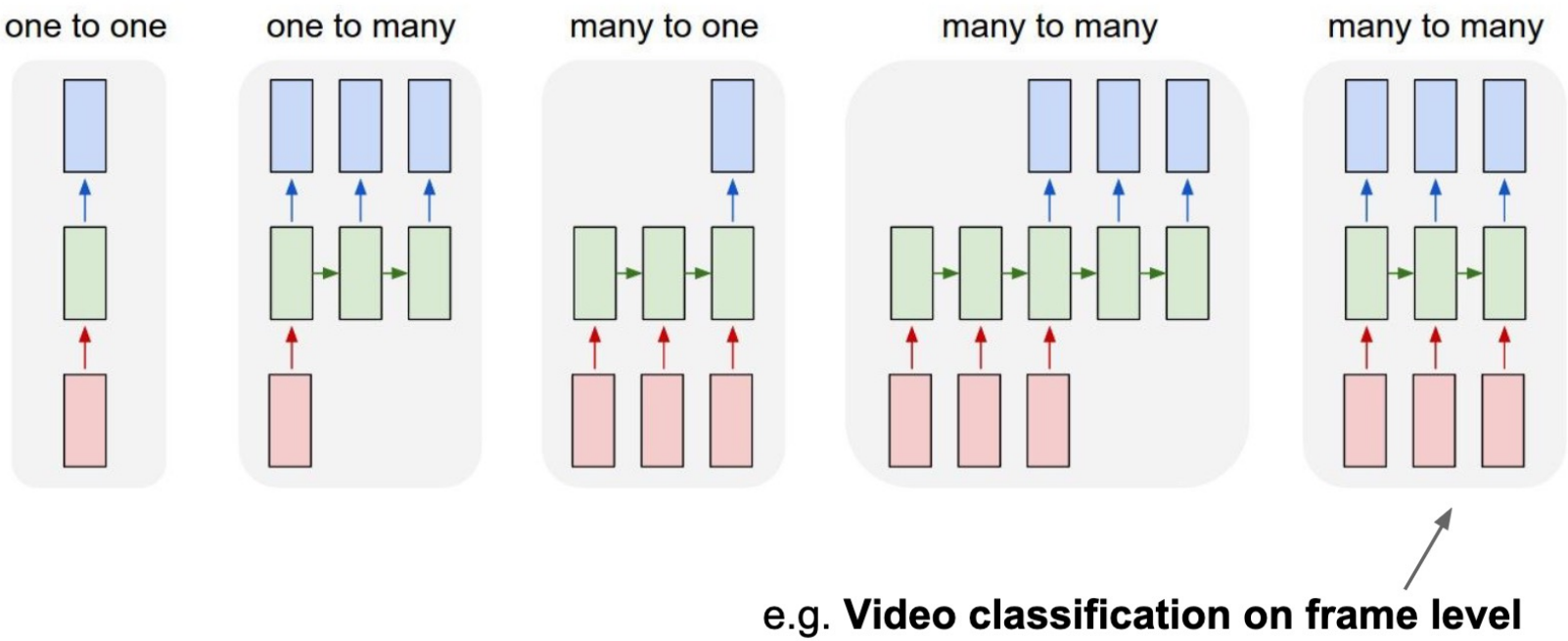
e.g. **Sentiment Classification**
sequence of words -> sentiment

Process Sequence



e.g. **Machine Translation (Chatbots)**
seq of words -> seq of words

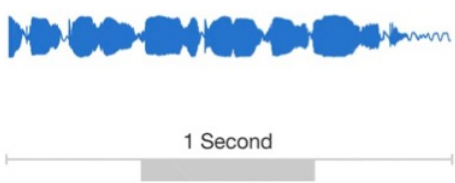
Process Sequence



Sequences Are Everywhere

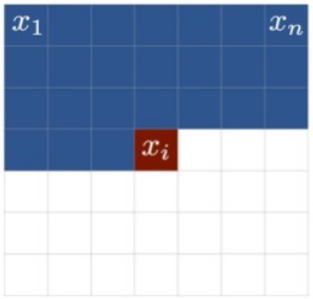
"Sequences really seem to be everywhere! We should learn how to model them. What is the best way to do that? Stay tuned!"

Words, letters



Speech

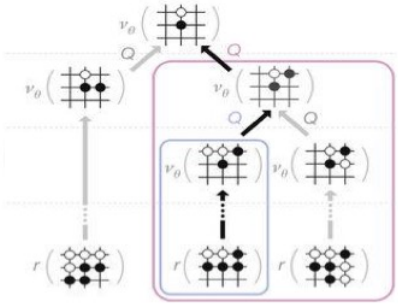
Videos



Images

```
import numpy as np
import matplotlib.pyplot as plt

def forward_backward_prop(w, T):
  hs = [0.5]
  for _ in range(T):
    hs.append(np.tanh(w*hs[-1]))

  dh = 1
  for t in range(T):
    dh = (1-hs[-1-t] ** 2) * w * dh

  return hs[-1], dh

T = 10  # sequence length
wlim = 4  #limit of interval over weights w

results = []
ws = np.linspace(-wlim, wlim, 1000)
for w in ws:
  results.append(forward_backward_prop(w, T))

plt.plot(ws, [r[0] for r in results], label='RNN state')
plt.plot(ws, [r[1] for r in results], label='Gradients')
```

Programs



Decision making

- Sequences are collections of variable length where order matters.
- Sequences are widespread across machine learning applications.
- Not all deep learning models can handle sequential data.

Training Machine Learning Models

	Supervised learning	Sequence modelling
Data	$\{x, y\}_i$	$\{x\}_i$
Model	$y \approx f_\theta(x)$	$p(x) \approx f_\theta(x)$
Loss	$\mathcal{L}(\theta) = \sum_{i=1}^{N} l(f_\theta(x_i), y_i)$	$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log p(f_\theta(x_i))$
Optimisation	$\theta^* = \arg\min_\theta \mathcal{L}(\theta)$	$\theta^* = \arg\max_\theta \mathcal{L}(\theta)$

Modeling p(x)

"Modeling word probabilities is really difficult"

Simplest model:
Assume independence of words

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$$

p("modeling") × p("word") × p("probabilities") × p("is") × p("really") × p("difficult")

Word	$p(x_i)$
the	0.049
be	0.028
...	...
really	0.0005
...	...

However:

Most likely 6-word sentence:

"The the the the the the."

→ Independence assumption does not match sequential structure of language.

Modeling p(x)

More realistic model:

Assume conditional dependence of words

$$p(x_T) = p(x_T | x_1, \ldots, x_{T-1})$$

Modeling word probabilities is really ?

Context | Target

Target	p(x|context)
difficult	0.01
hard	0.009
fun	0.005
...	...
easy	0.00001

Modeling p(x)

The chain rule

Computing the joint p(**x**) from conditionals

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t | x_1, ..., x_{t-1})$$

Modeling

Modeling **word**

Modeling word **probabilities**

Modeling word probabilities **is**

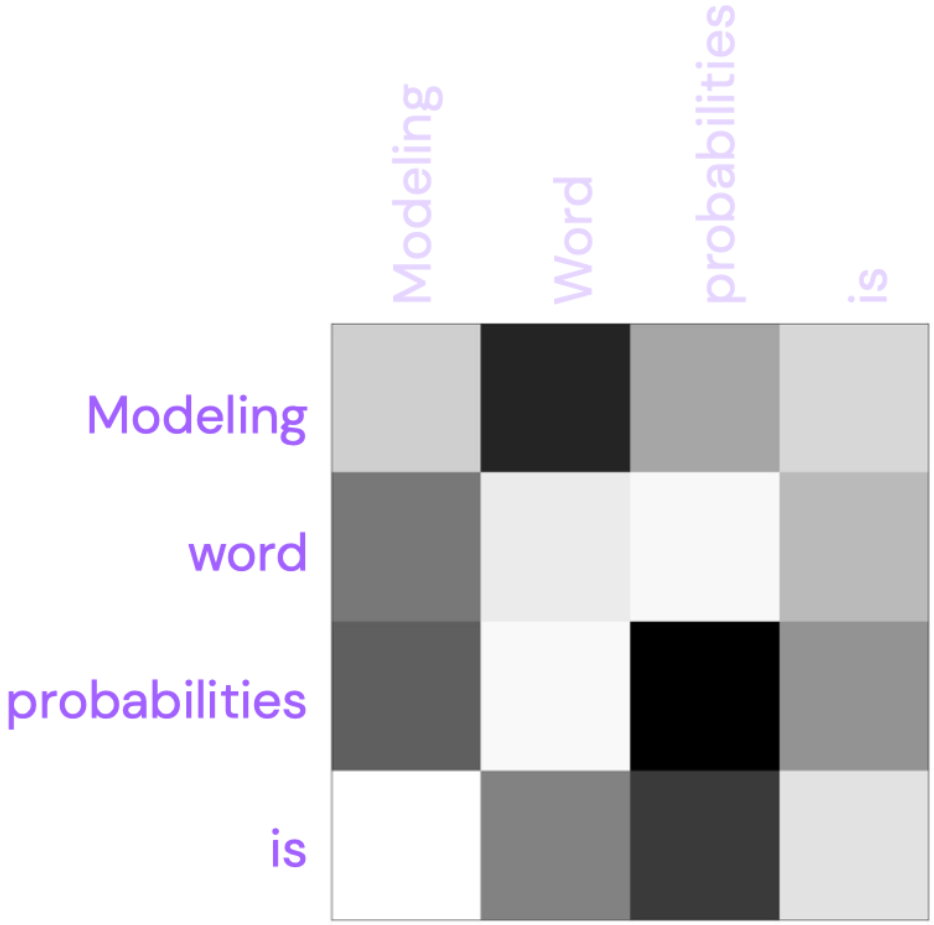
Modeling word probabilities is **really**

Modeling word probabilities is really **difficult**

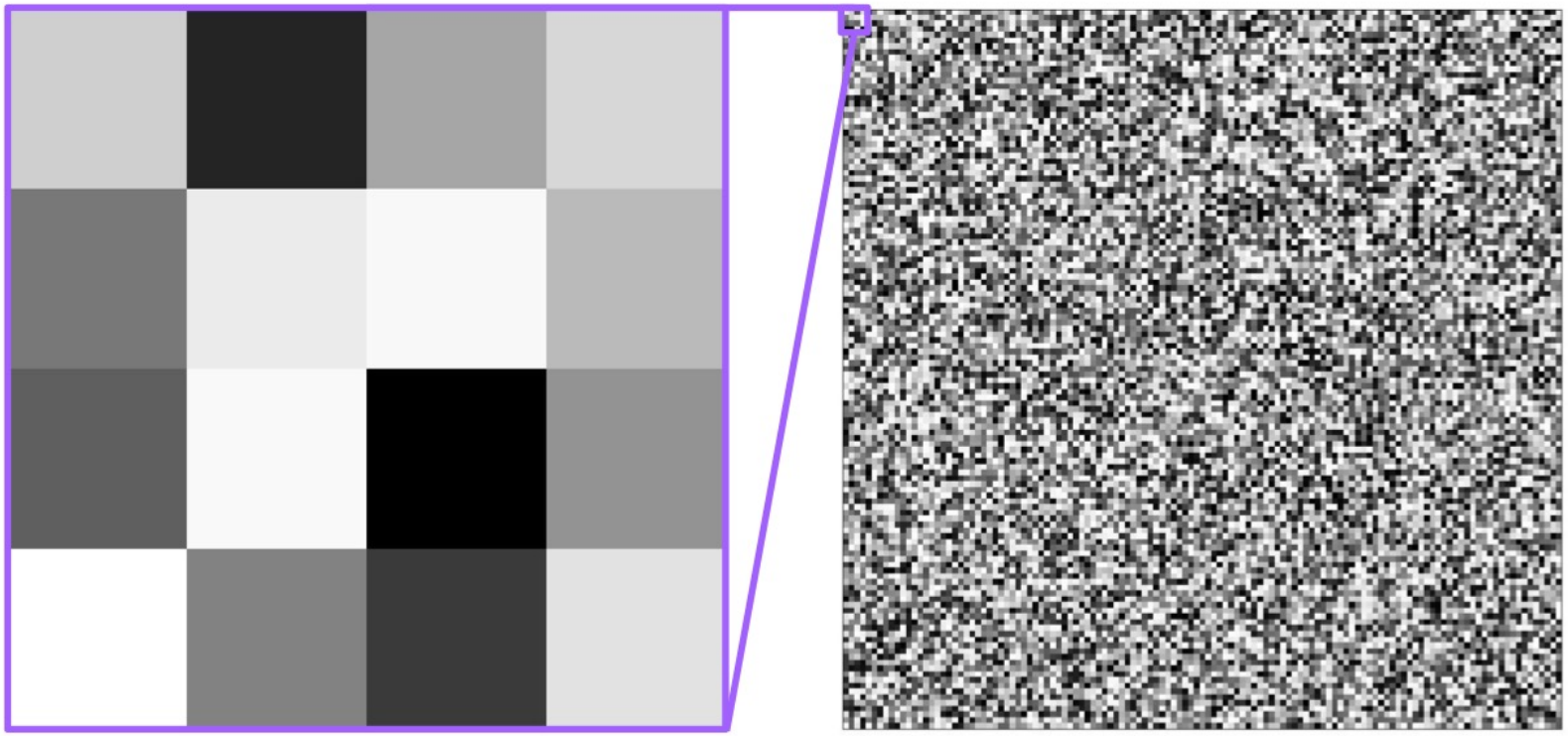
$$p(x_1)$$
$$p(x_2|x_1)$$
$$p(x_3|x_2, x_1)$$
$$p(x_4|x_3, x_2, x_1)$$
$$p(x_5|x_4, x_3, x_2, x_1)$$
$$p(x_6|x_5, x_4, x_3, x_2, x_1)$$

Scalability Issues

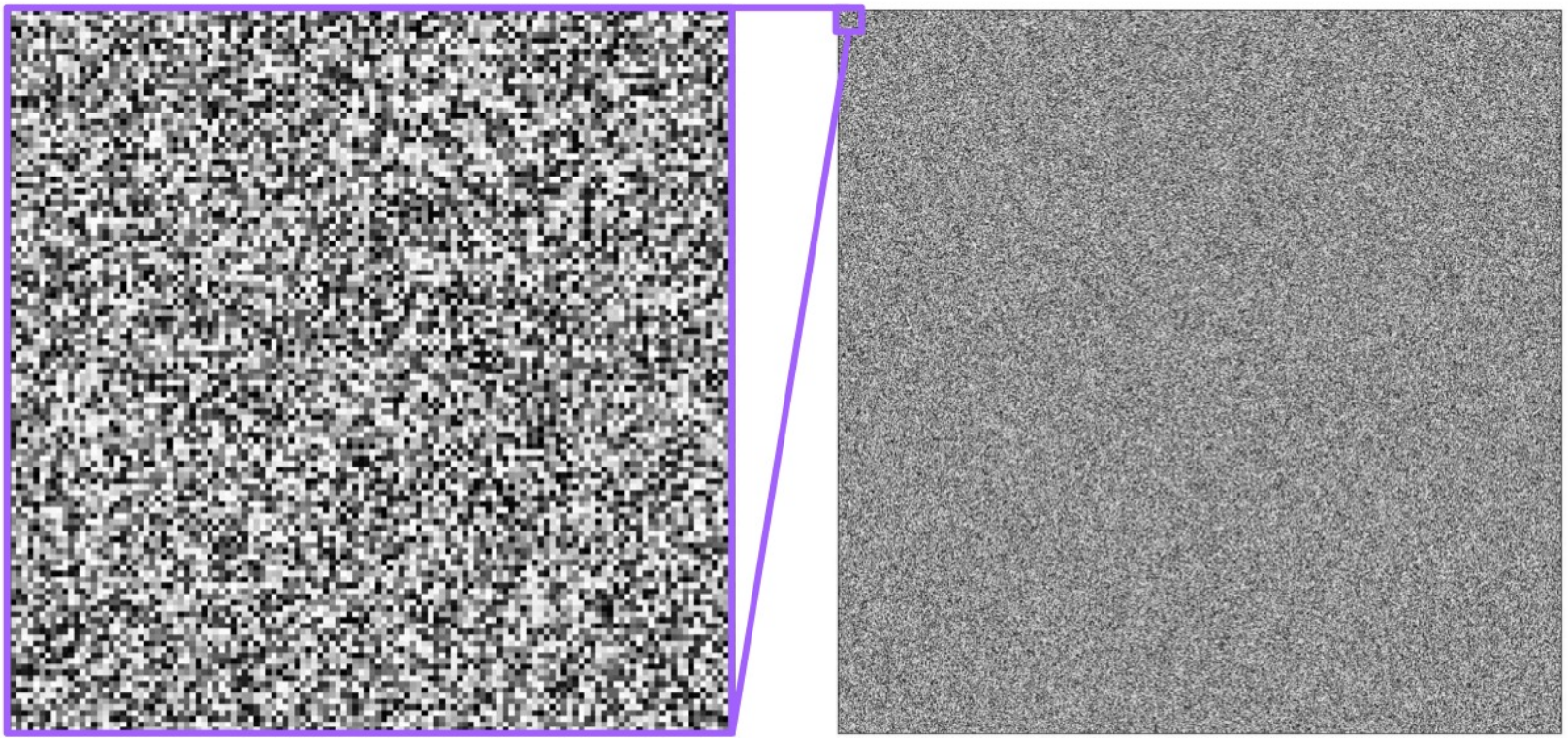
$$p(x_2|x_1)$$



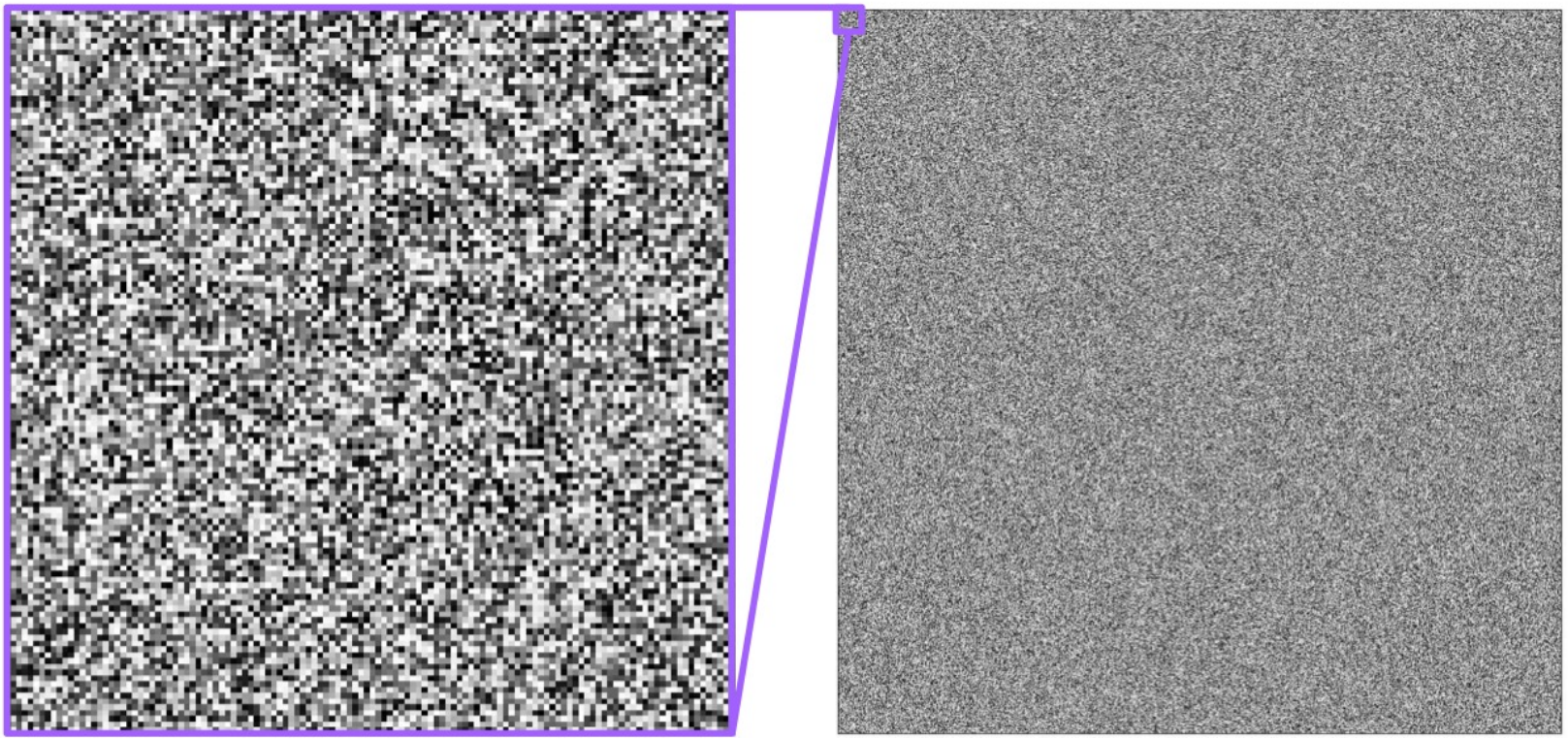
Scalability Issues



Scalability Issues



Scalability Issues



These images are only for context of size N=1!
The table size of larger contexts will grow with **vocabulary**^{N}

Fixing a small context: N-grams

Only condition on N previous words

$$p(\mathbf{x}) \approx \prod_{t=1}^{T} p(x_t | x_{t-N-1}, \ldots, x_{t-1})$$

Modeling

Modeling **word**

Modeling word **probabilities**

word probabilities **is**

probabilities is **really**

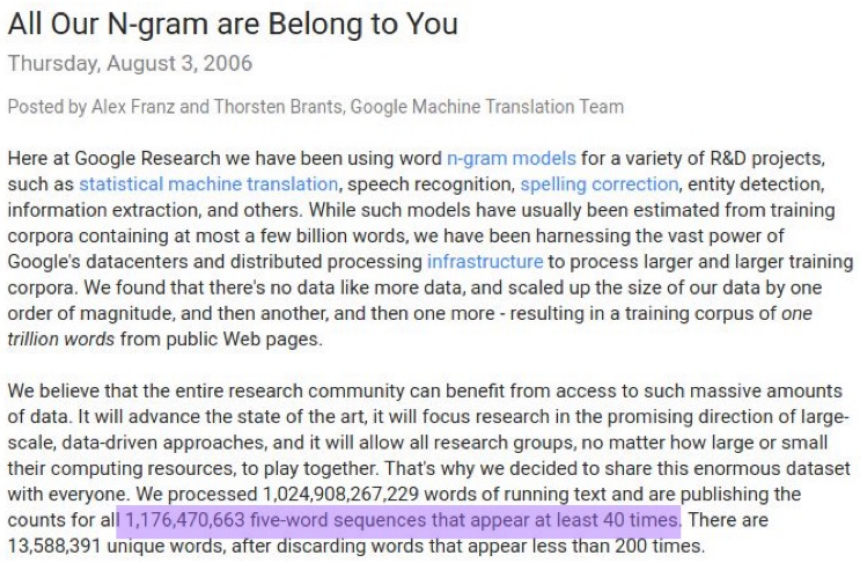
is really **difficult**

$$p(x_1)$$
$$p(x_2|x_1)$$
$$p(x_3|x_2, x_1)$$
$$p(x_4|x_3, x_2)$$
$$p(x_5|x_4, x_3)$$
$$p(x_6|x_5, x_4)$$

Downsides of Using N-grams

1. Doesn't take into account words that are more than N words away

2. Data table is still very, very large



All Our N-gram are Belong to You

Thursday, August 3, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of *one trillion words* from public Web pages.

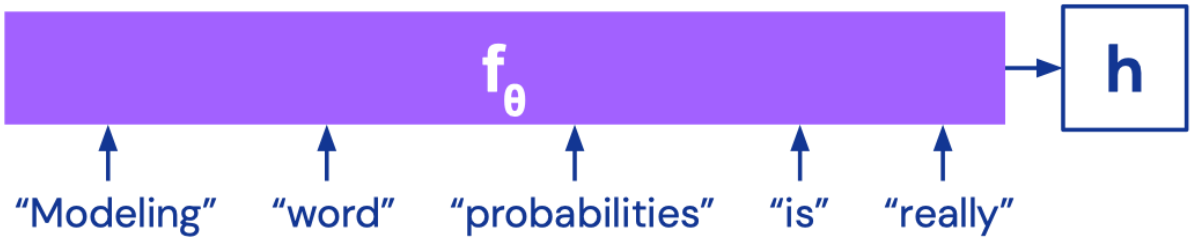
We believe that the entire research community can benefit from access to such massive amounts of data. It will advance the state of the art, it will focus research in the promising direction of large-scale, data-driven approaches, and it will allow all research groups, no matter how large or small their computing resources, to play together. That's why we decided to share this enormous dataset with everyone. We processed 1,024,908,267,229 words of running text and are publishing the counts for all 1,176,470,663 five-word sequences that appear at least 40 times. There are 13,588,391 unique words, after discarding words that appear less than 200 times.

- Modeling probabilities of sequences scales badly given the non-independent structure of their elements.

Learning to Model Word Probabilities

- Can this probability estimation be learned from data in a more efficient way?

1. Vectorising the context



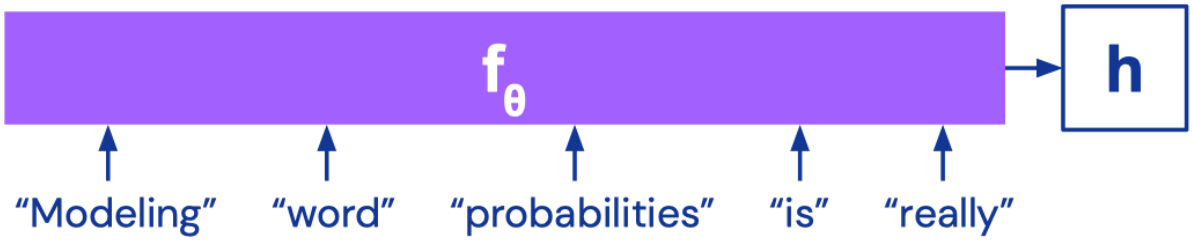
f_θ summarises the context in **h** such that:

$$p(x_t|x_1, \ldots, x_{t-1}) \approx p(x_t|h)$$

Learning to Model Word Probabilities

- Can this probability estimation be learned from data in a more efficient way?

1. Vectorising the context



Desirable properties for f_θ:

Order matters

Variable length

Differentiable

Pairwise encoding

Preserves long-term

N-grams

$$p(\mathbf{x}) \approx \prod_{t=1}^{T} p(x_t | x_{t-N-1}, ..., x_{t-1})$$

Modeling $p(x_1)$

Modeling **word** $p(x_2|x_1)$

Modeling word **probabilities** $p(x_3|x_2, x_1)$

word probabilities **is** $p(x_4|x_3, x_2)$

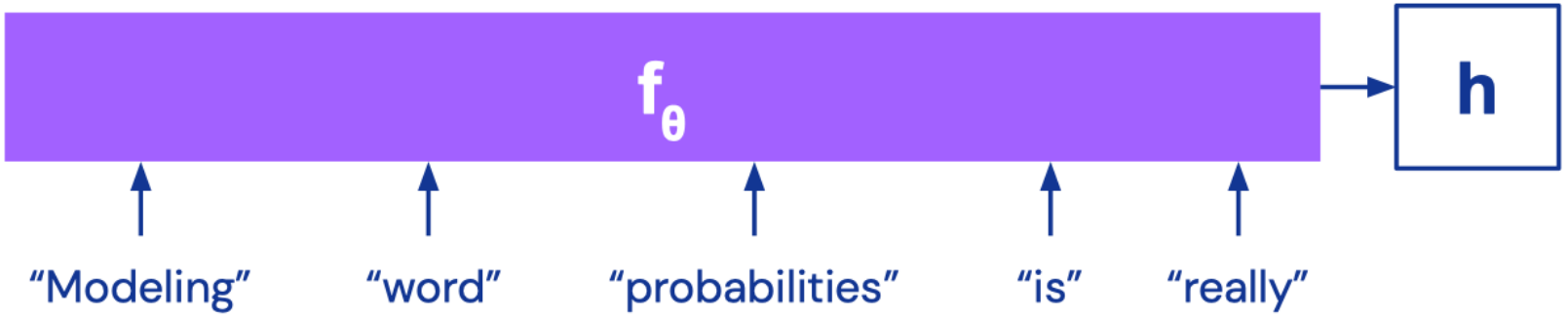
probabilities is **really** $p(x_5|x_4, x_3)$

is really **difficult** $p(x_6|x_5, x_4)$

f_θ concatenates the N last words

	N-gram
Order matters	✔
Variable length	✘
Differentiable	✘
Pairwise encoding	✔
Preserves long-term	✘

Addition



	N-gram	Addition
Order matters	✓	✗
Variable length	✗	✓
Differentiable	✗	✓
Pairwise encoding	✓	✗
Preserves long-term	✗	✓

- N-grams and simple aggregation do not meet the requirements for modeling sequences.

Learning to Model Word Probabilities

Modeling conditional probabilities

Desirable properties for g_θ :

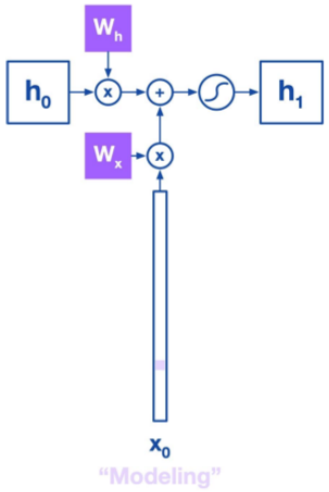
Individual changes can have large effects (non-linear/deep)

Returns a probability distribution

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs)

Persistent state variable **h** stores information from the context observed so far.



$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)$$

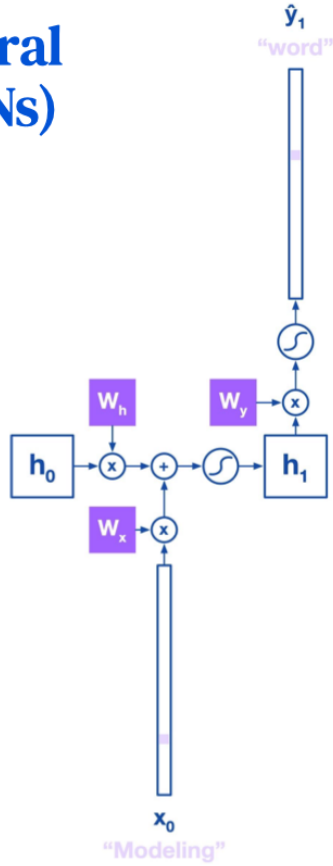
Word representation

Words are indexed and represented as 1-hot vectors

Large Vocabulary of possible words $|V|$

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs)



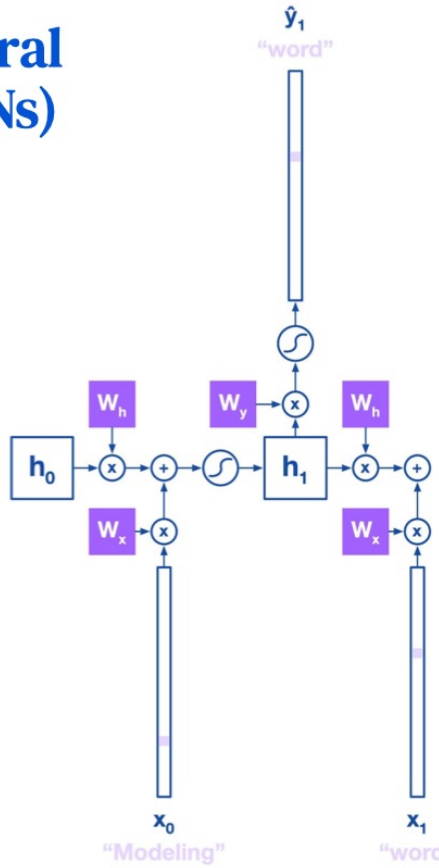
RNNs predict the target **y** (the next word) from the state **h.**

$$p(\mathbf{y_{t+1}}) = softmax(\mathbf{W}_y \mathbf{h}_t)$$

Softmax ensures we obtain a distribution over all possible words.

Recurrent Neural Networks (RNNs)

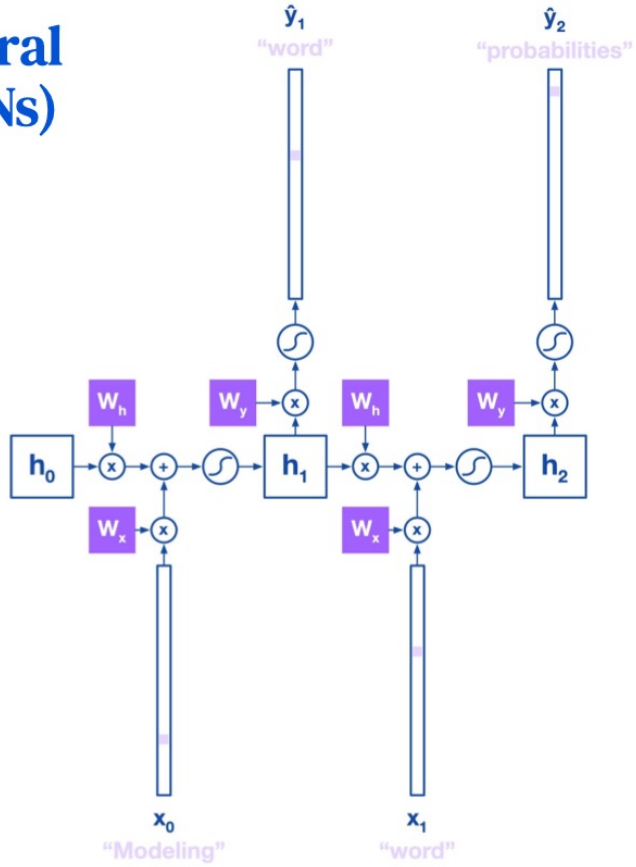
Recurrent Neural Networks (RNNs)



Input next word in sentence x_1

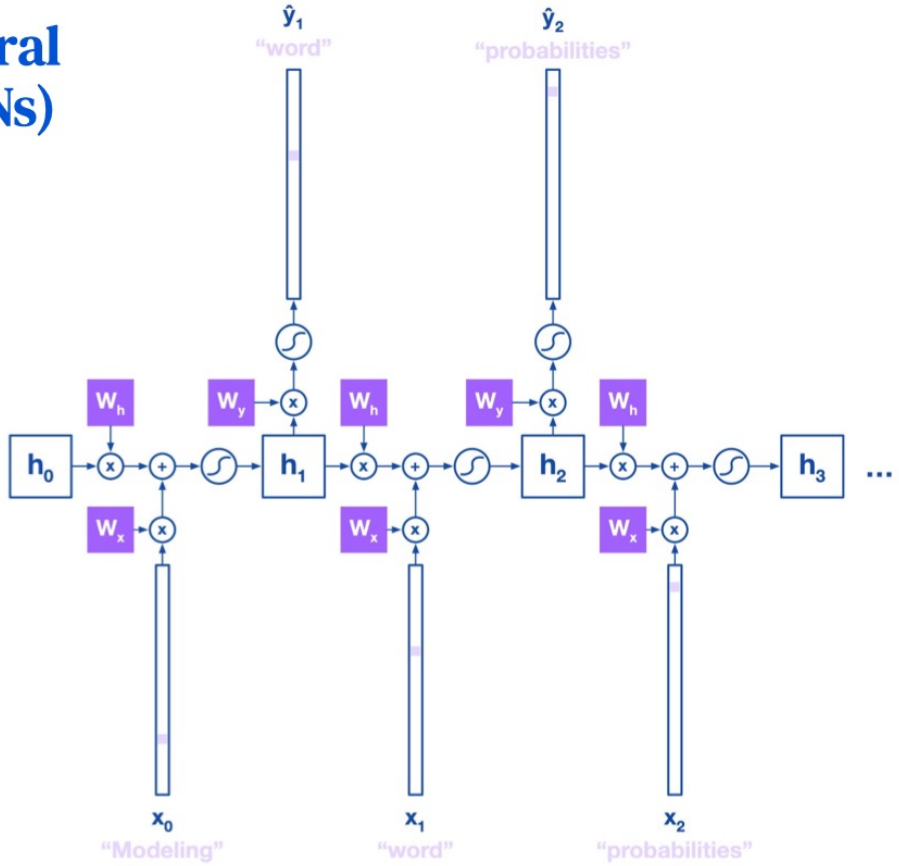
Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs)



Recurrent Neural Networks (RNNs)

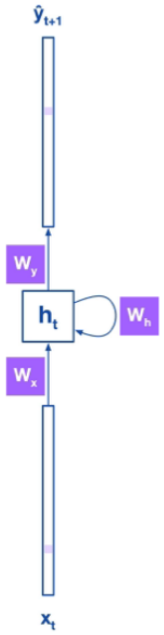
Recurrent Neural Networks (RNNs)



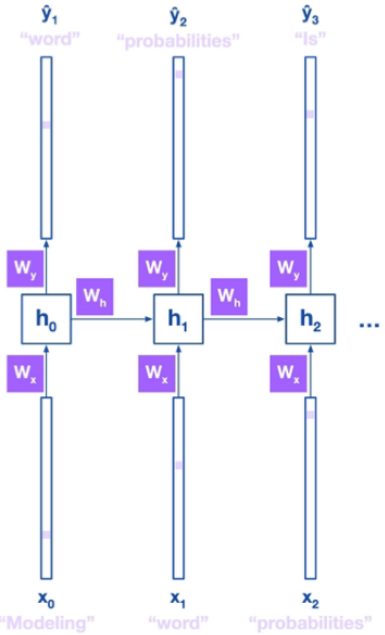
Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs)

Weights are shared over time steps



RNN



RNN rolled out over time

Loss: Cross Entropy

Next word prediction is essentially a classification task where the number of classes is the size of the vocabulary.

As such we use the cross-entropy loss:

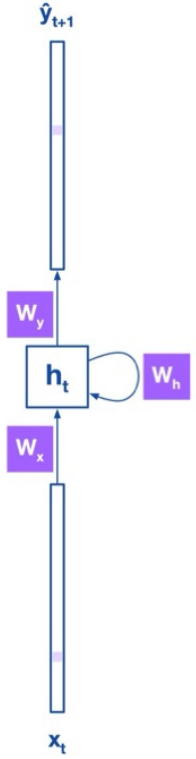
For one word:

$$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t = -\mathbf{y}_t \log \hat{\mathbf{y}}_t$$

For the sentence:

$$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{t=1}^{T} \mathbf{y}_t \log \hat{\mathbf{y}}_t$$

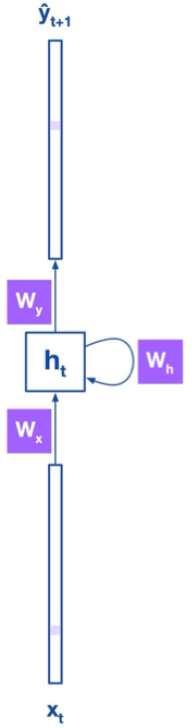
With parameters $\theta = \{\mathbf{W}_y, \mathbf{W}_x, \mathbf{W}_h\}$



Differentiating

Parameters: $\mathbf{W}_y$, $\mathbf{W}_x$ and $\mathbf{W}_h$

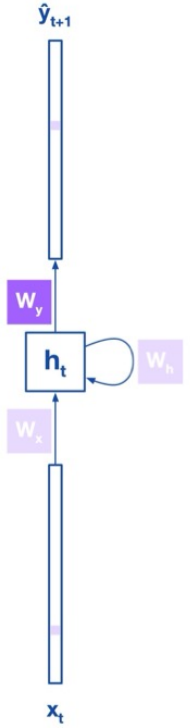
$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)$$
$$p(\mathbf{x}_{t+1}) = softmax(\mathbf{W}_y \mathbf{h}_t)$$
$$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t = -\mathbf{y}_t \log \hat{\mathbf{y}}_t$$



Differentiating

Parameters: $\mathbf{W}_y$, $\mathbf{W}_x$ and $\mathbf{W}_h$

$$\mathbf{h}_t = \tanh(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}_x\mathbf{x}_t)$$
$$p(\mathbf{x}_{t+1}) = softmax(\mathbf{W}_y\mathbf{h}_t)$$
$$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t = -\mathbf{y}_t \log \hat{\mathbf{y}}_t$$



$\mathbf{W}_y$:

$$\frac{\partial \mathcal{L}_{\theta,t}}{\partial \mathbf{W}_y} = \frac{\partial \mathcal{L}_{\theta,t}}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{W}_y}$$
$$= (\mathbf{y}_t - \hat{\mathbf{y}}_t)\mathbf{h}_t$$

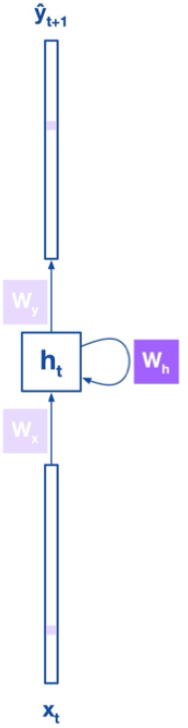
Differentiating

Parameters: $\mathbf{W_y}$, $\mathbf{W_x}$ and $\mathbf{W_h}$

$$\mathbf{h}_t = \tanh(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}_x\mathbf{x}_t)$$
$$p(\mathbf{x}_{t+1}) = softmax(\mathbf{W}_y\mathbf{h}_t)$$
$$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t = -\mathbf{y}_t \log \hat{\mathbf{y}}_t$$

$\mathbf{W_h}$:

$$\frac{\partial \mathcal{L}_{\theta,t}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}_{\theta,t}}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h}$$



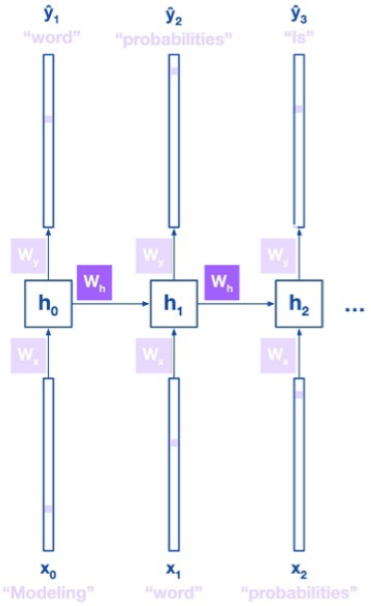
Differentiating

Parameters: $\mathbf{W_y}$, $\mathbf{W_x}$ and $\mathbf{W_h}$

$$\mathbf{h}_t = \tanh(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}_x\mathbf{x}_t)$$
$$p(\mathbf{x}_{t+1}) = softmax(\mathbf{W}_y\mathbf{h}_t)$$
$$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t = -\mathbf{y}_t \log \hat{\mathbf{y}}_t$$

$\mathbf{W_h}$:

$$\frac{\partial \mathcal{L}_{\theta,t}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}_{\theta,t}}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h}$$



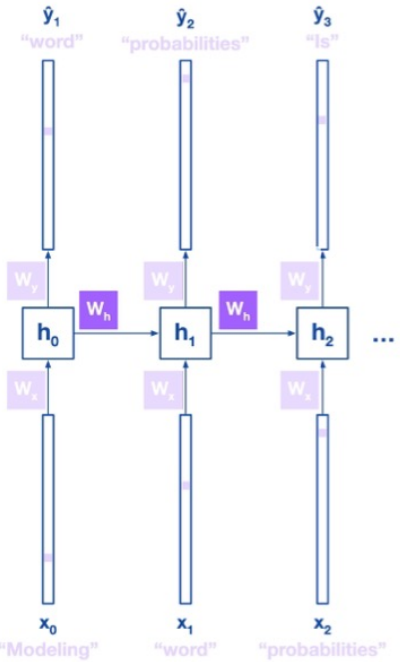
Differentiating

Back propagating through time

$$
\begin{aligned}
\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h} &= \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_h} \\
&= \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \left[\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_h} + \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}} \frac{\partial \mathbf{h}_{t-2}}{\partial \mathbf{W}_h} \right] \\
&\dots \\
&= \sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_h}
\end{aligned}
$$

Differentiating

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)$$
$$p(\mathbf{x}_{t+1}) = softmax(\mathbf{W}_y \mathbf{h}_t)$$
$$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t = -\mathbf{y}_t \log \hat{\mathbf{y}}_t$$



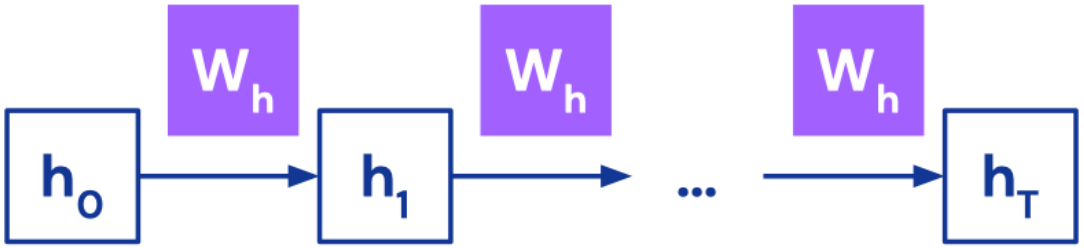
$$\frac{\partial \mathcal{L}_{\theta,t}}{\partial \mathbf{W}_h} = \frac{\partial \mathcal{L}_{\theta,t}}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h}$$

$$= \sum_{k=1}^{t} \frac{\partial \mathcal{L}_{\theta,t}}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_h}$$

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h} = \sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_h}$$

Vanishing Gradients

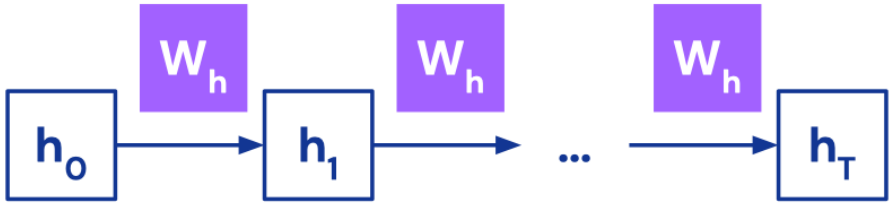
A simple example



$$\mathbf{h}_t = \mathbf{W}_h \mathbf{h}_{t-1} \qquad \mathbf{h}_t \to \infty \text{ if } |\mathbf{W}_h| > 1$$
$$\mathbf{h}_t = (\mathbf{W}_h)^t \mathbf{h}_0 \qquad \mathbf{h}_t \to 0 \text{ if } |\mathbf{W}_h| < 1$$

Vanishing Gradients

A simple example



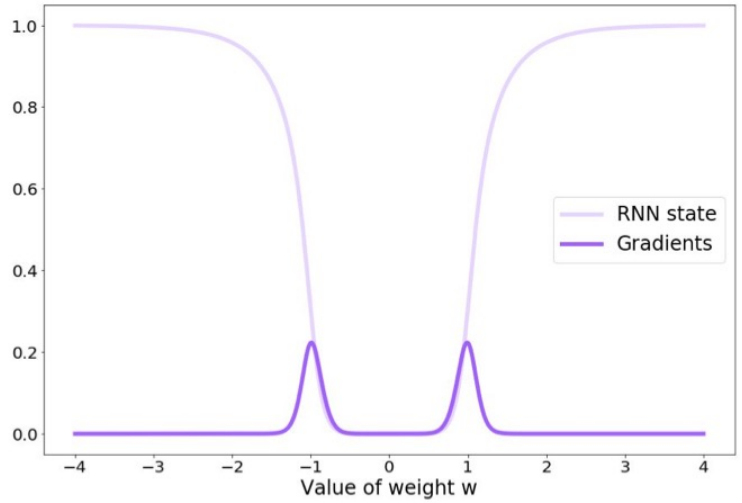
But RNNs bound h with a tanh!

$$\mathbf{h}_t = \mathbf{W}_h \mathbf{h}_{t-1} \rightarrow \mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1})$$

Vanishing Gradients

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1})$$

```python
import numpy as np
import matplotlib.pyplot as plt

def forward_backward_prop(w, T):
  hs = [0.5]
  for _ in range(T):
    hs.append(np.tanh(w*hs[-1]))

  dh = 1
  for t in range(T):
    dh = (1-hs[-1-t] ** 2) * w * dh

  return hs[-1], dh

T = 10  # sequence length
wlim = 4  #limit of interval over weights w

results = []
ws = np.linspace(-wlim, wlim, 1000)
for w in ws:
  results.append(forward_backward_prop(w, T))

plt.plot(ws, [r[0] for r in results], label='RNN state')
plt.plot(ws, [r[1] for r in results], label='Gradients')
```



$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-T}} = (1 - \tanh^2(\mathbf{W}_h \mathbf{h}_{t-1}))\mathbf{W}_h \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-T}}$$

Properties of RNNs

	N-gram	Addition	RNN
Order matters	✓	✗	✓
Variable length	✗	✓	✓
Differentiable	✗	✓	✓
Pairwise encoding	✓	✗	✗
Preserves long-term	✗	✓	✗

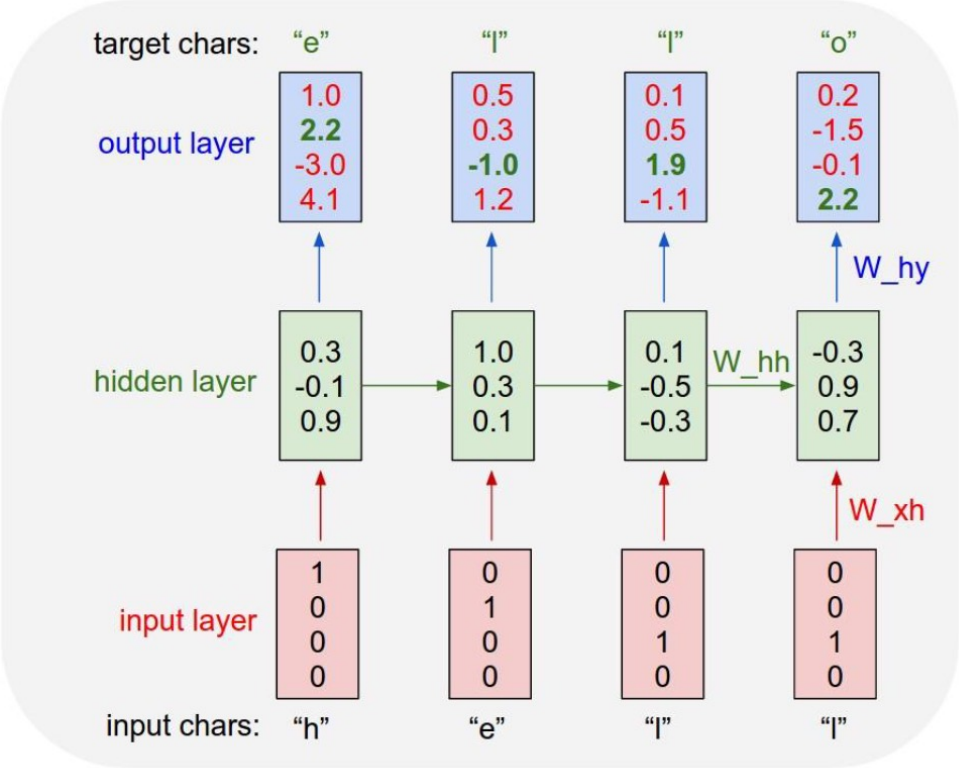
- Recurrent neural networks can model sequences of variable length and can be trained via back-propagation.
- They do, however suffer from the vanishing gradients problem, which stops them from capturing long-term dependencies.

Example: Character-level Language Model

Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

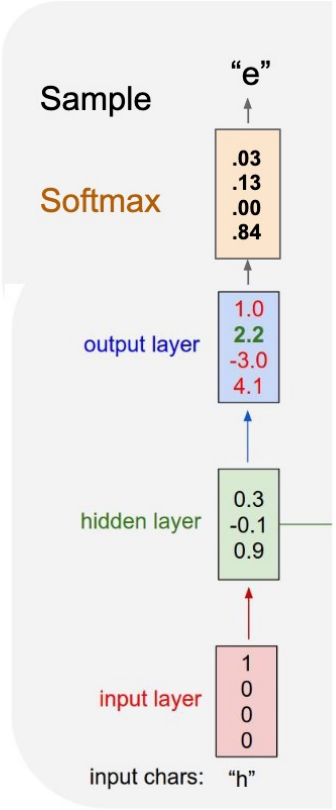


Example: Character-level Language Model

Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

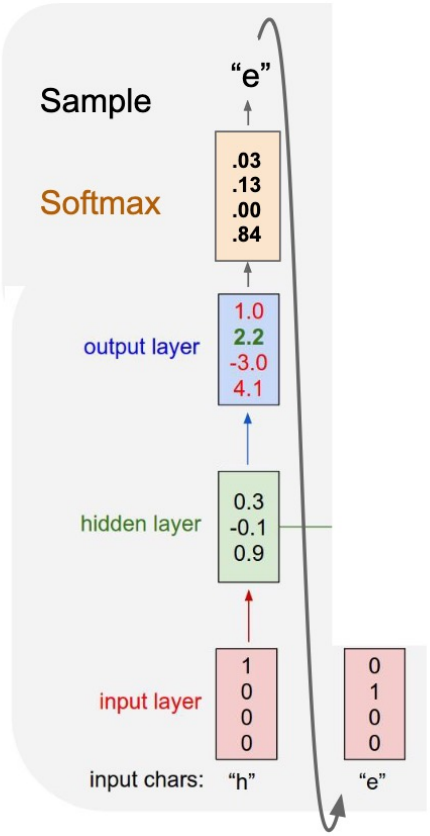


Example: Character-level Language Model

Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

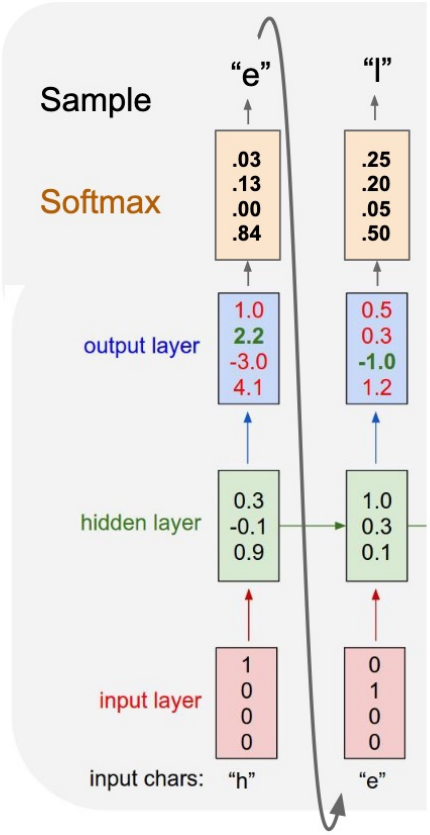


Example: Character-level Language Model

Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

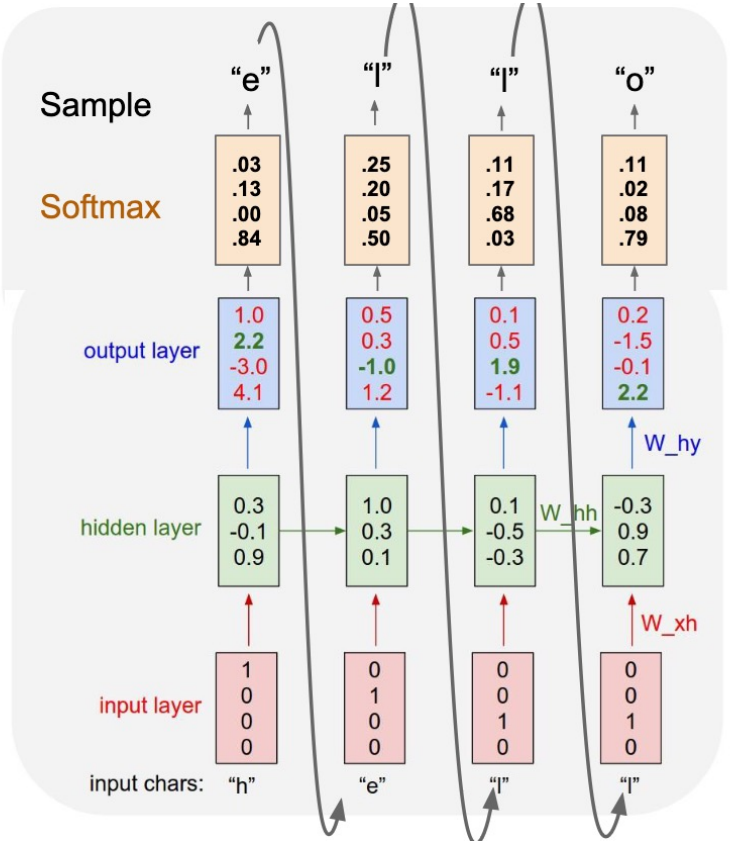


Example: Character-level Language Model

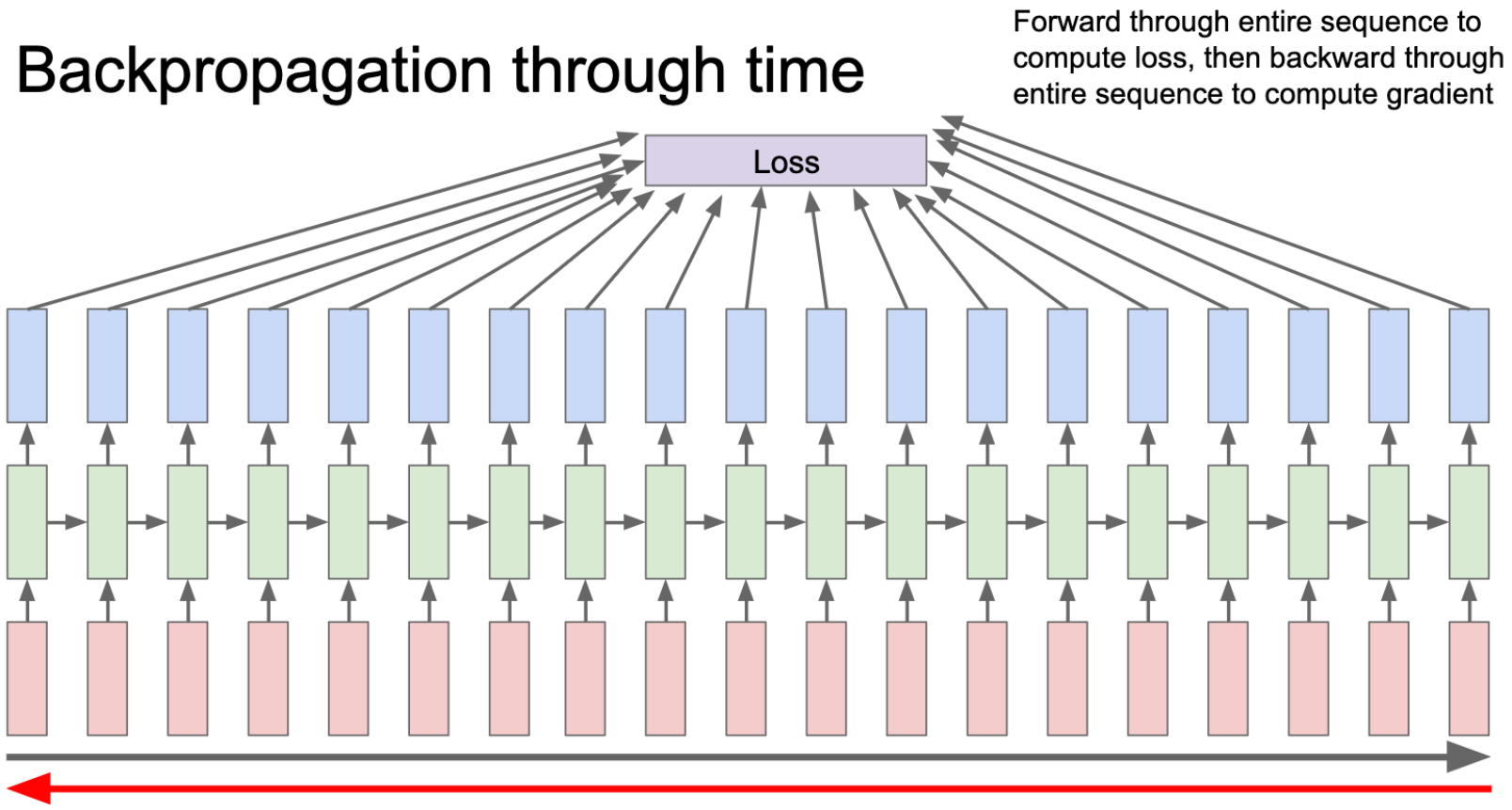
Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

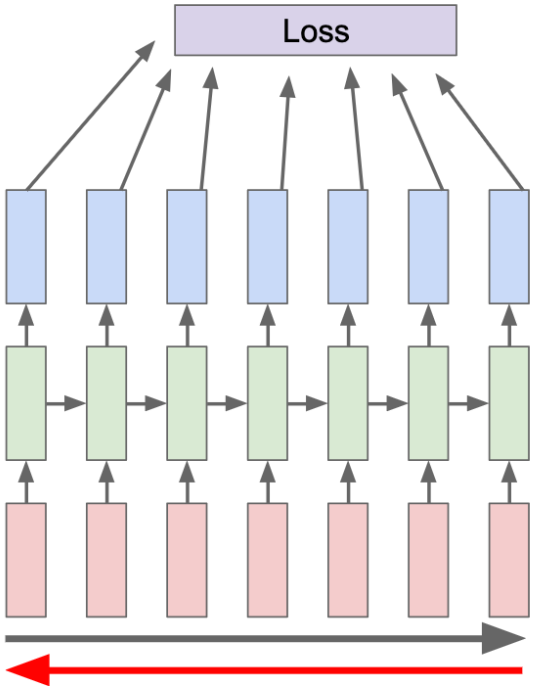


Backpropagation Through Time



Truncated Backpropagation Through Time

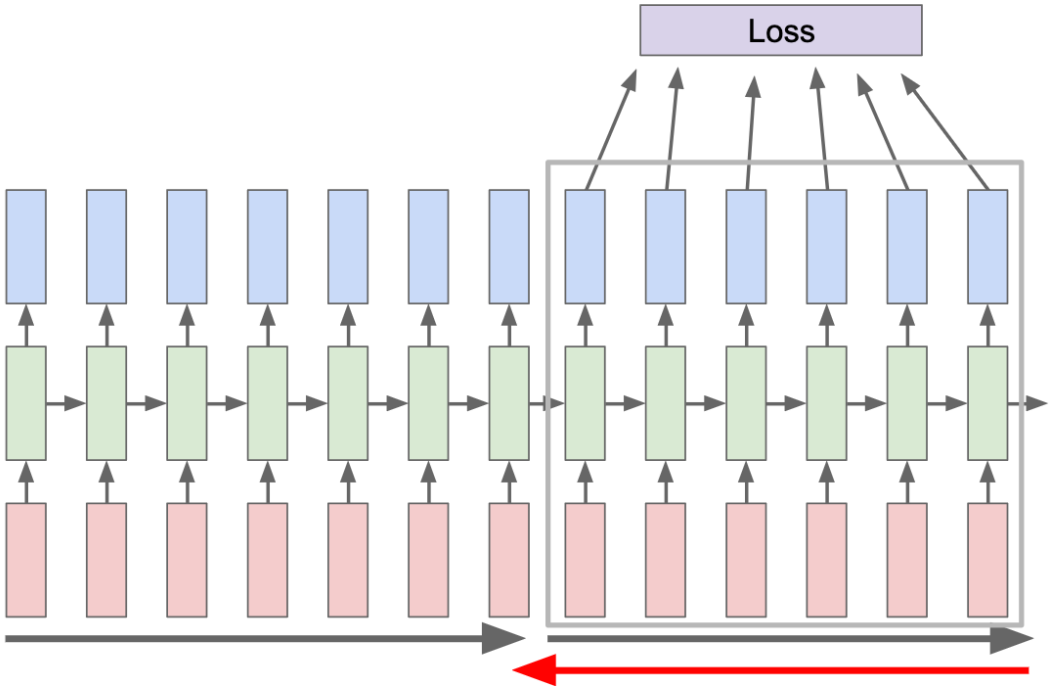
Truncated Backpropagation through time



Run forward and backward through chunks of the sequence instead of whole sequence

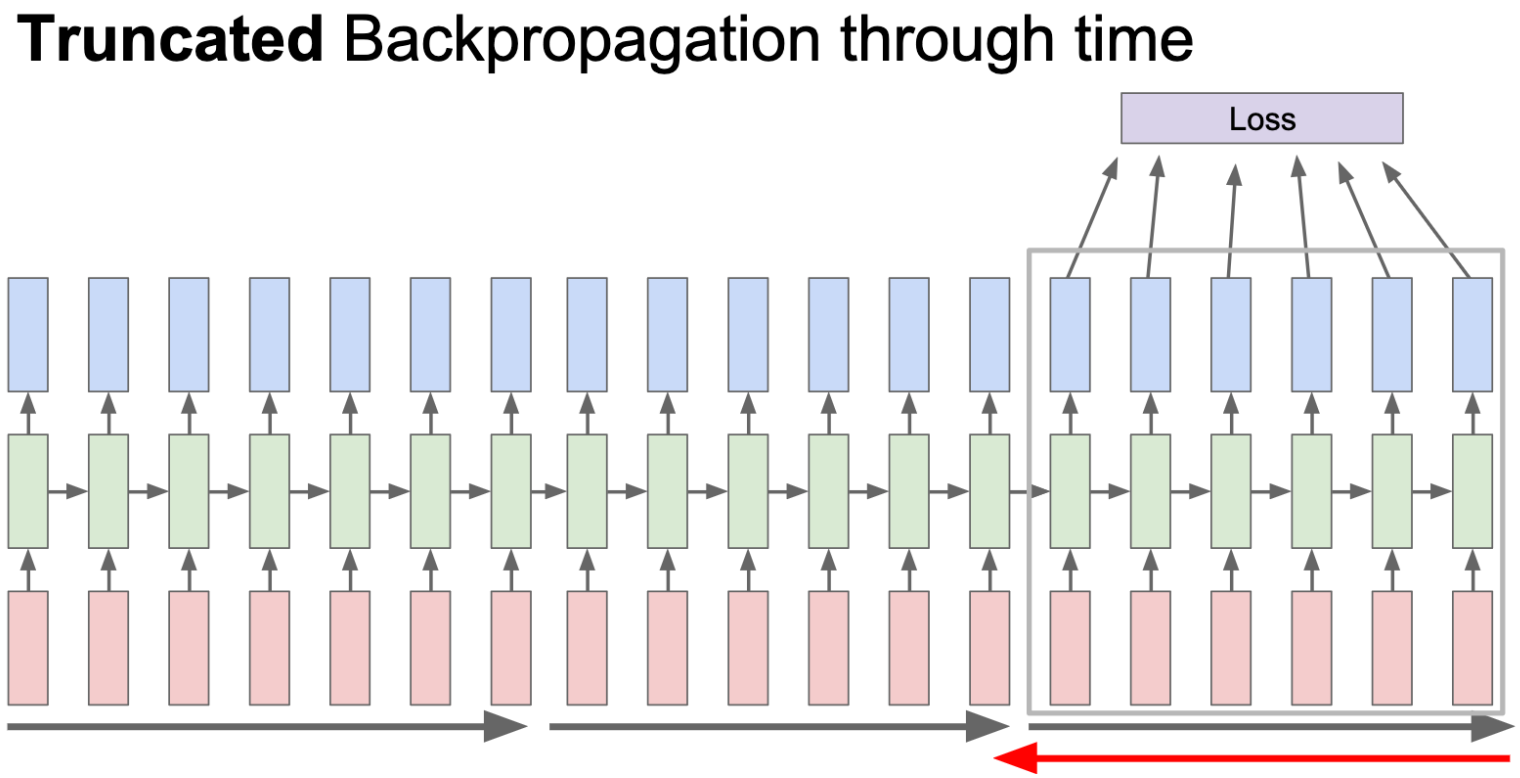
Truncated Backpropagation Through Time

Truncated Backpropagation through time



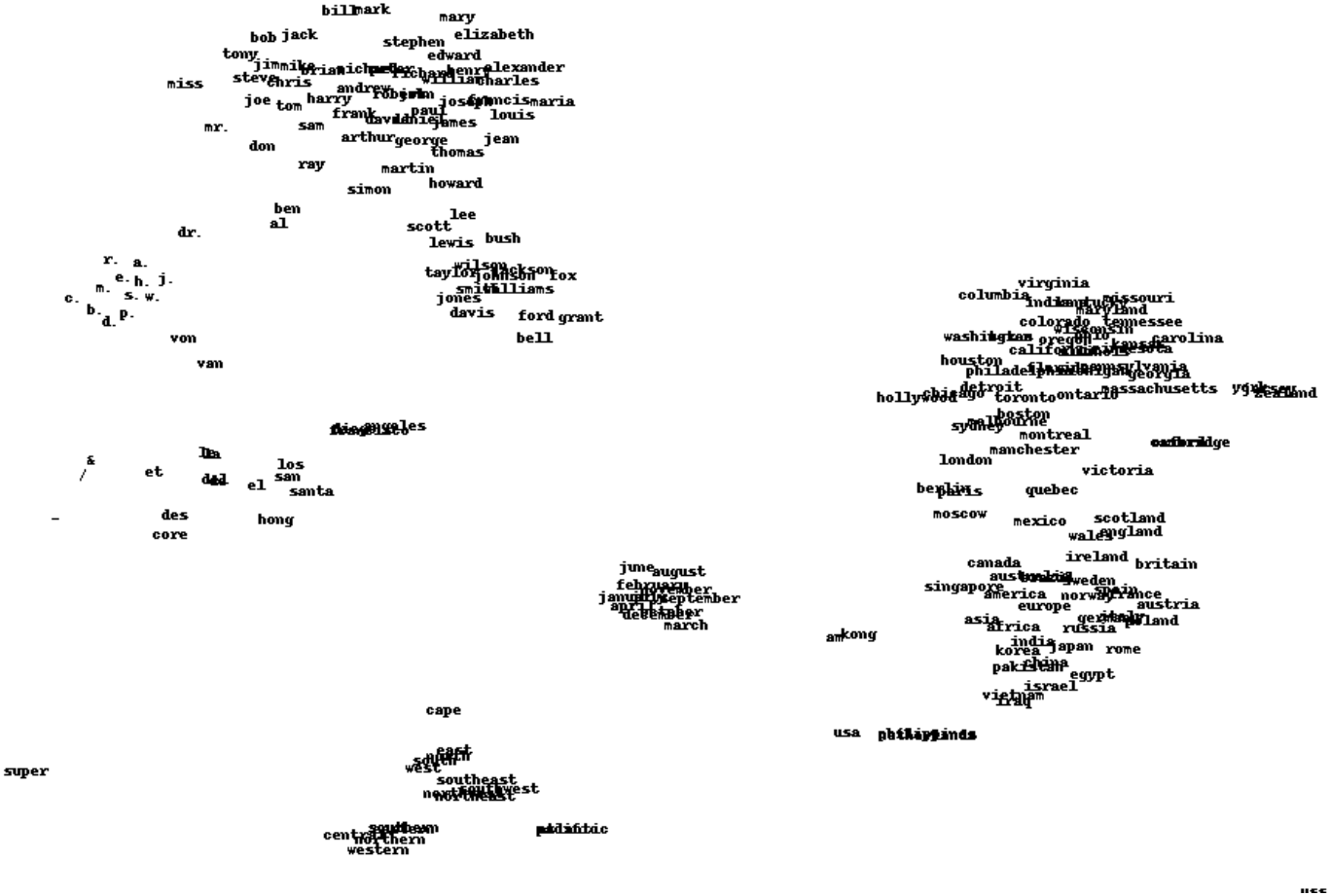
Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

Truncated Backpropagation Through Time

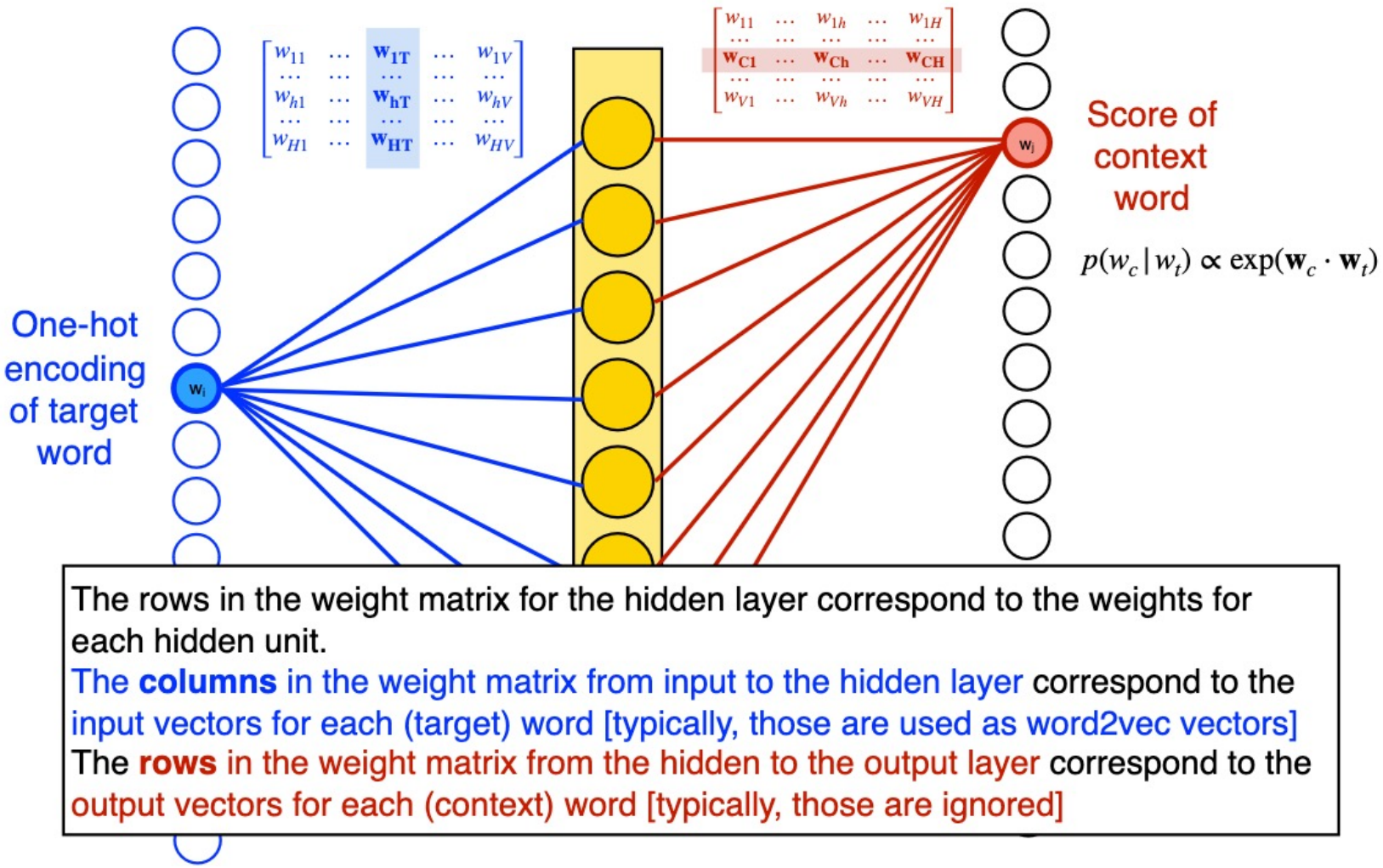


Word Embedding

- Can we represent words as vectors in space?



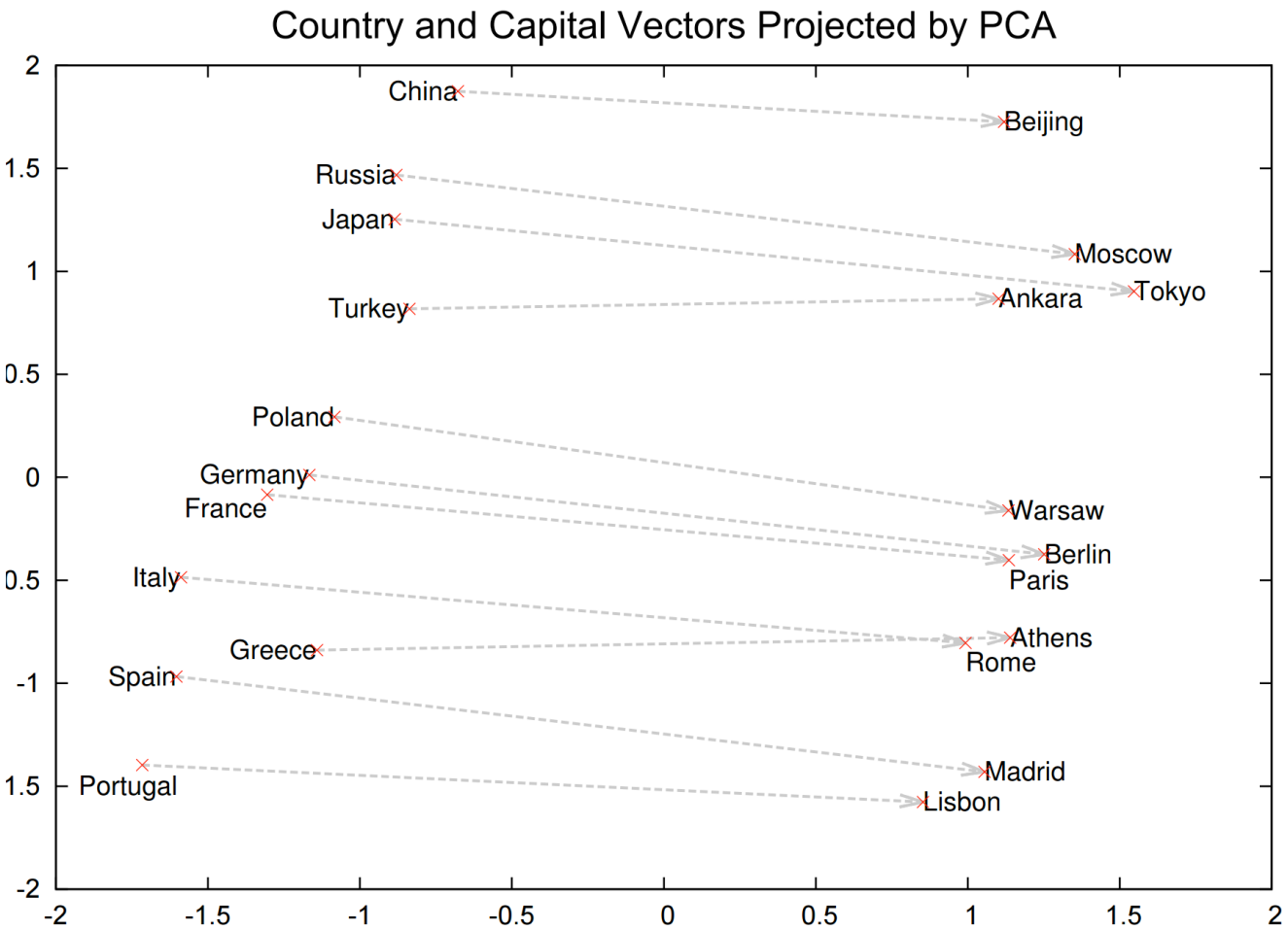
Word Embedding



The rows in the weight matrix for the hidden layer correspond to the weights for each hidden unit.
The **columns** in the weight matrix from input to the hidden layer correspond to the input vectors for each (target) word [typically, those are used as word2vec vectors]
The **rows** in the weight matrix from the hidden to the output layer correspond to the output vectors for each (context) word [typically, those are ignored]

Word Embedding

- Word Analogies:



Questions?

UNIVERSITY OF NORTH CAROLINA CHARLOTTE