Exploring the Naturalness of Code with Recurrent Neural Networks

Jack Lanchantin and Ji Gao

April 26, 2016

University of Virginia

Background/Motivation

“The cat sat apple the mat.” → High entropy (unnatural)

“The cat sat on the mat.” → Low entropy (natural)

```
def convolve_features(int a, int b) → ??
    ...
```

Hypothesis: “unnatural” code is suspicious, possibly suggesting a bug

Language Models

$$P(S) = P(t_1, t_2, \ldots t_N) = P(t_1) \cdot \prod_{i=2}^{N} \underbrace{P(t_i | t_1, \ldots, t_{i-1})}_{\text{Hard to compute}}$$

n-gram Language Models

$$P(S) = P(t_1, t_2, \ldots t_N) = P(t_1) \cdot \prod_{i=2}^{N} \underline{P(t_i | t_1, \ldots, t_{i-1})}$$

$$P_{n\text{gram}}(t_i | h) = P(t_i | t_{i-n+1}, \ldots, t_{i-1})$$

Entropy (information theory)

Entropy (H) measures the amount of uncertainty in a distribution

$$H = -\sum p(x) \log p(x)$$

English text has between 0.6 and 1.3 bits of entropy for each character

Goal: Find the entropy of a new line from a previously trained language model in order to determine if it is a possibly buggy line

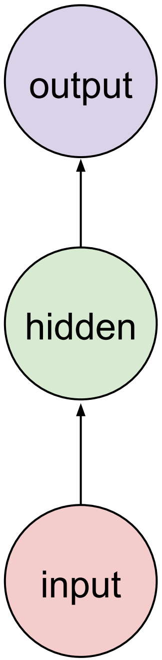
Our Project

Goal: Test more complex language models and see if we can better predict buggy lines based on entropy

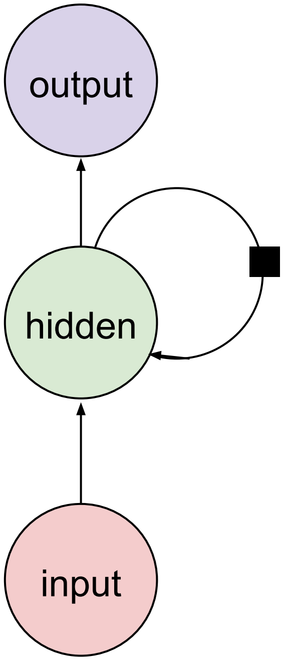
Method: We use Recurrent Neural Networks as our language model which can more accurately handle long term dependencies in the language

Recurrent Neural Networks (RNNs)

Traditional Neural Network

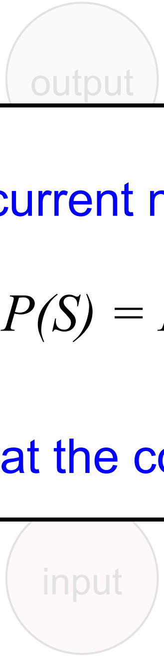


Recurrent Neural Network

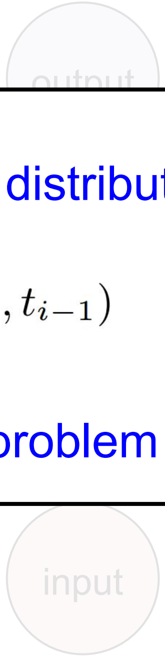


Recurrent Neural Networks (RNNs)

Traditional Neural Network



Recurrent Neural Network

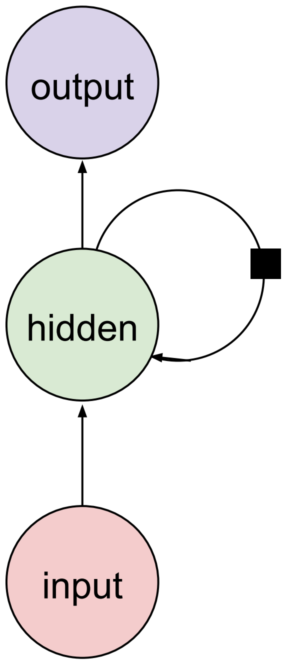


Recurrent nets can model the ***full*** conditional distribution

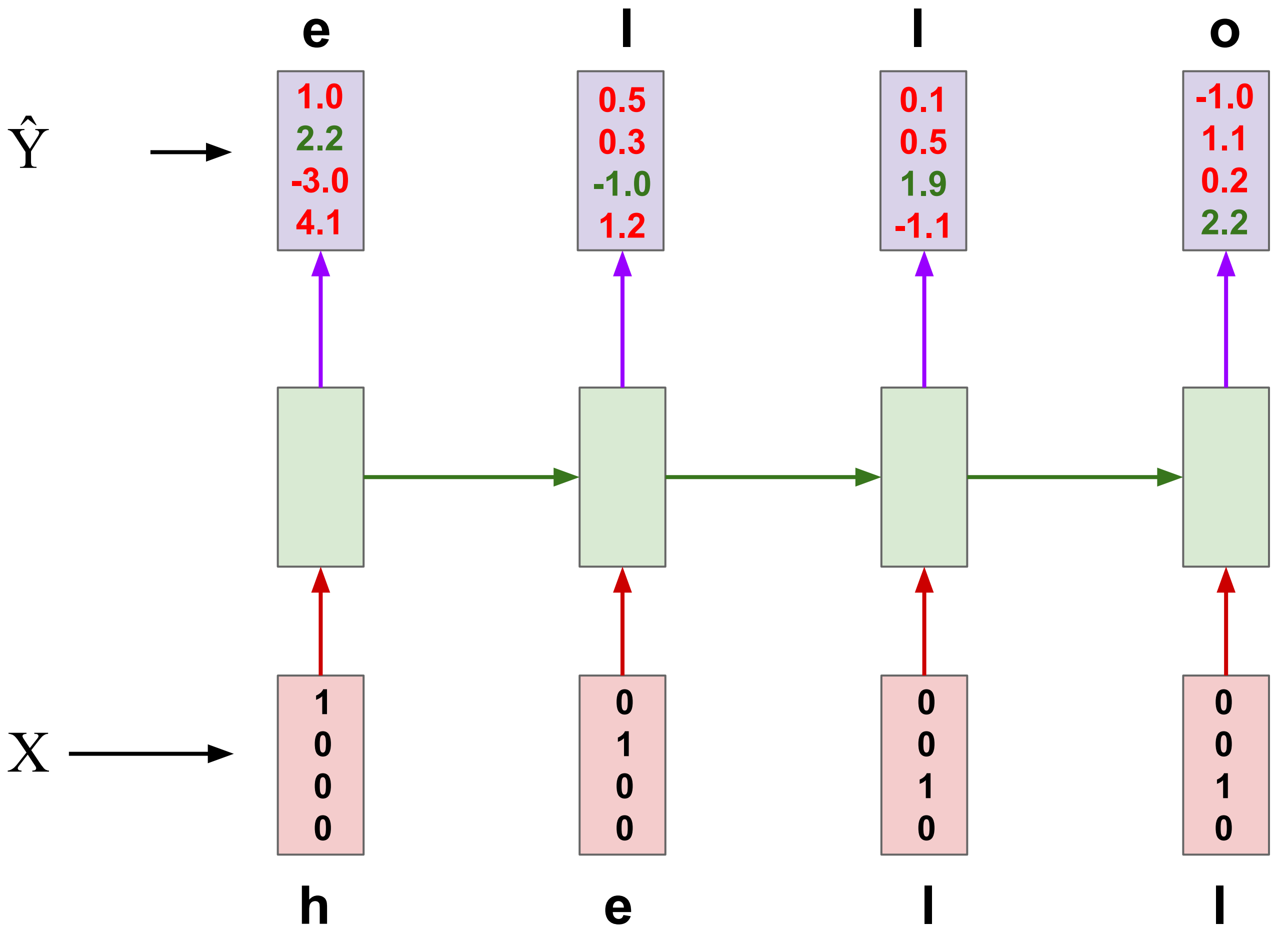
$$P(S) = P(t_1, t_2, \ldots t_N) = P(t_1) \cdot \prod_{i=2}^{N} P(t_i | t_1, \ldots, t_{i-1})$$

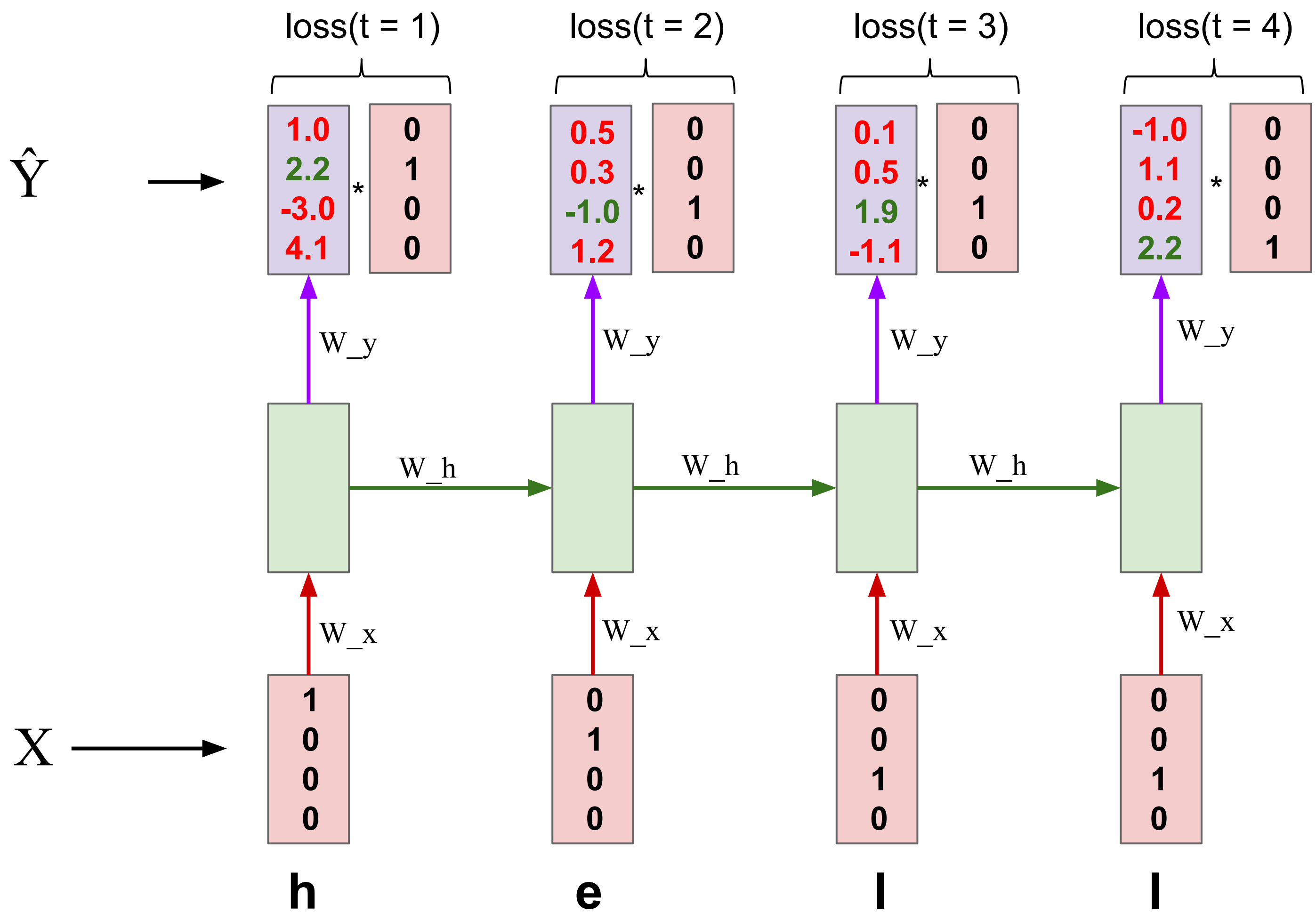
… at the cost of a much higher optimization problem

Character-level RNN Language Model

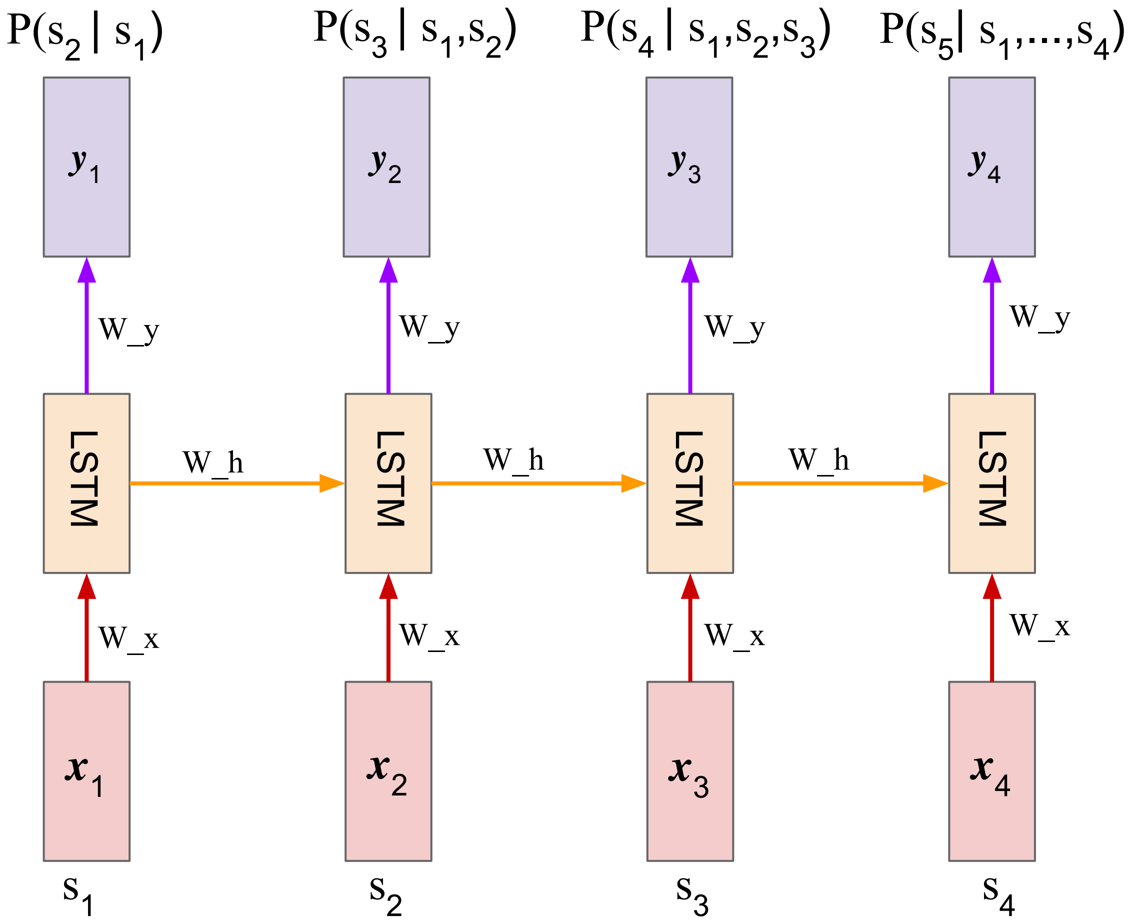


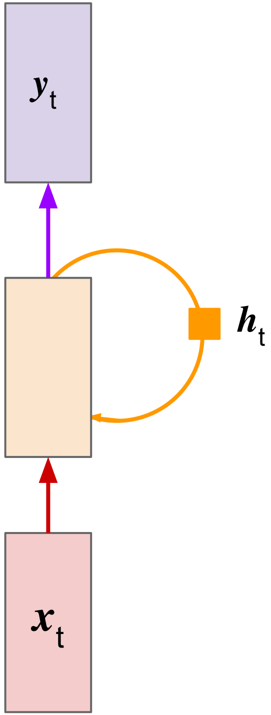
Character-level RNN Language Model



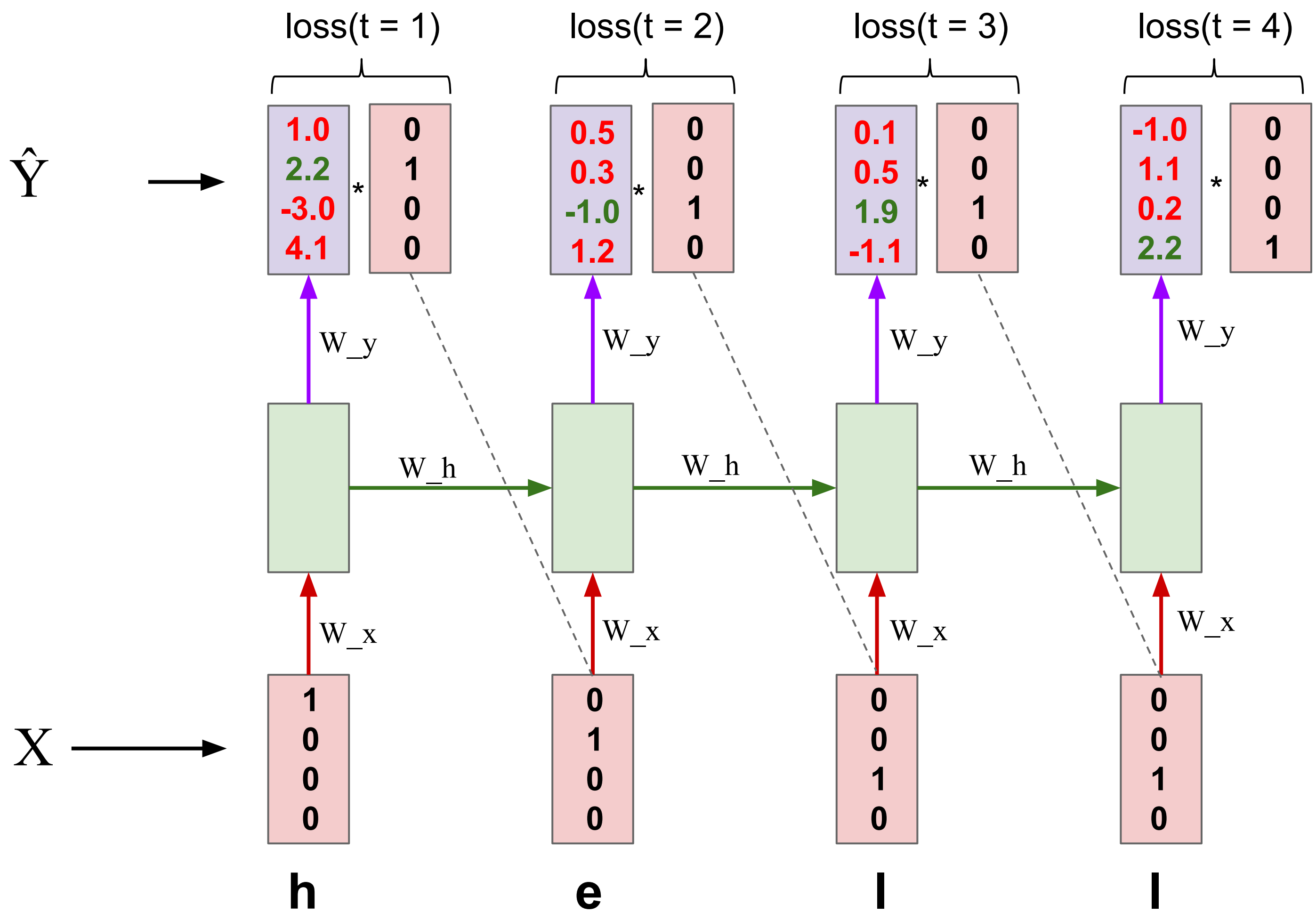


loss(t = 1)
loss(t = 2)
loss(t = 3)
loss(t = 4)
Ŷ
1.0
2.2
-3.0
4.1
*
0
1
0
0
0.5
0.3
-1.0
1.2
*
0
0
1
0
0.1
0.5
1.9
-1.1
*
0
0
1
0
-1.0
1.1
0.2
2.2
*
0
0
0
1
W_y
W_h
W_x
X
1
0
0
0
0
1
0
0
0
0
1
0
0
0
1
0
h
e
l
l


P(s2 | s1)
P(s3 | s1,s2)
P(s4 | s1,s2,s3)
P(s5| s1,...,s4)
y1
y2
y3
y4
W_y
W_y
W_y
W_y
LSTM
LSTM
LSTM
LSTM
W_h
W_h
W_h
W_x
W_x
W_x
W_x
x1
x2
x3
x4
s1
s2
s3
s4



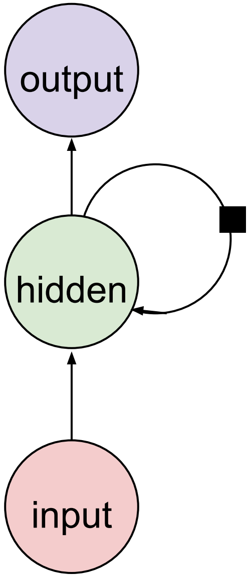
y_t
h_t
x_t


loss(t = 1)
loss(t = 2)
loss(t = 3)
loss(t = 4)
Ŷ
1.0 2.2 -3.0 4.1
*
0 1 0 0
0.5 0.3 -1.0 1.2
*
0 0 1 0
0.1 0.5 1.9 -1.1
*
0 0 1 0
-1.0 1.1 0.2 2.2
*
0 0 0 1
W_y
W_y
W_y
W_y
W_h
W_h
W_h
W_x
W_x
W_x
W_x
X
1 0 0 0
0 1 0 0
0 0 1 0
0 0 1 0
h
e
l
l

Long Short Term Memory Networks (LSTMs)

Recurrent networks suffer from the “vanishing gradient problem”

- Aren’t able to model long term dependencies in sequences

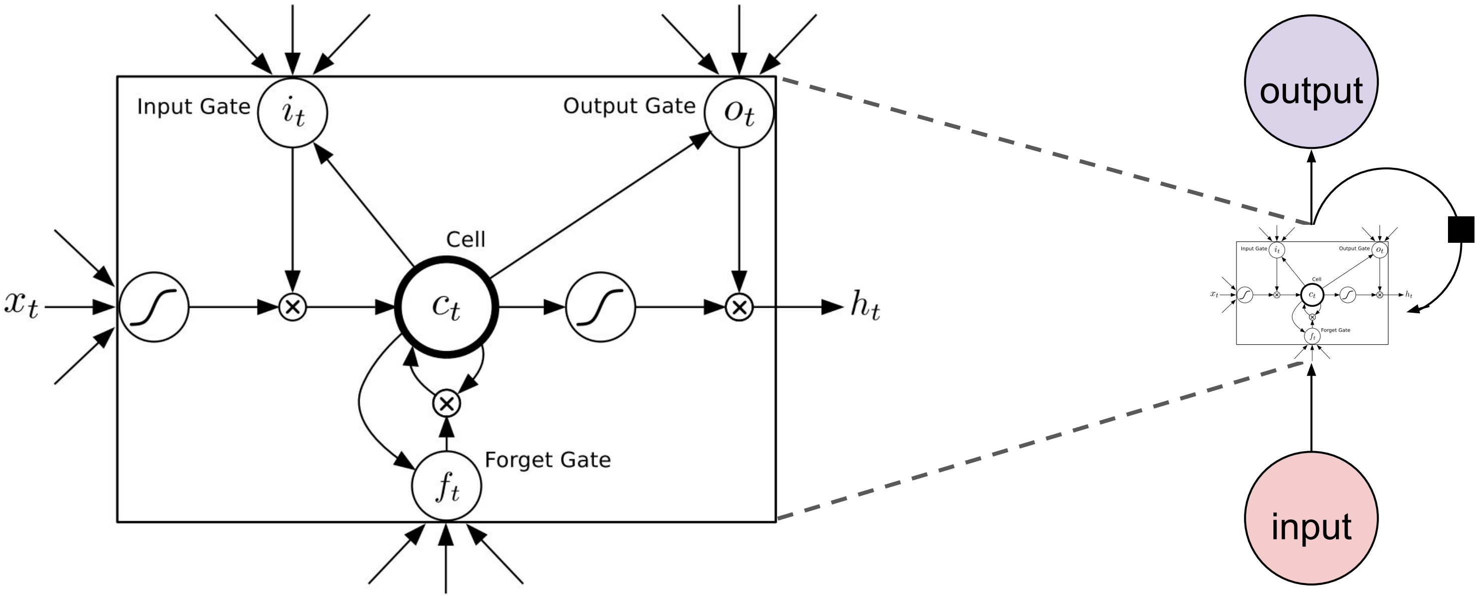


Long Short Term Memory Networks (LSTMs)

Recurrent networks suffer from the “vanishing gradient problem”

- Aren’t able to model long term dependencies in sequences

Use “gating units” to learn when to remember



Model design

We train two models:

- **Global language model (GLM)**
- **Local language model (LLM)**

To get the final entropy, we evaluate

- the entropy from the global model
- the combined entropy of the local and global model:

$$H_{total} = \lambda H_{GLM} + (1-\lambda) H_{LLM}$$

Dataset

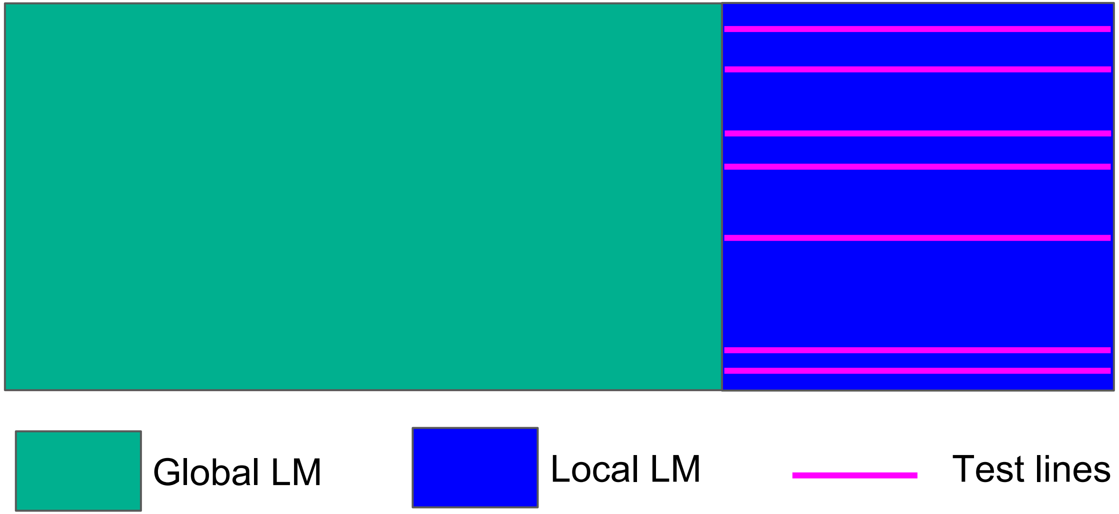
Elasticsearch project on Github

	Snapshots	JAVA files	Lines
Training set	50	118,164	16,502,732
Testing set*	18	59,180	9,437,902
Total	68	177,344	25,940,634

*To actually test, we selected 10,902 buggy lines and 10,902 non-buggy lines from the total 9,437,902 lines

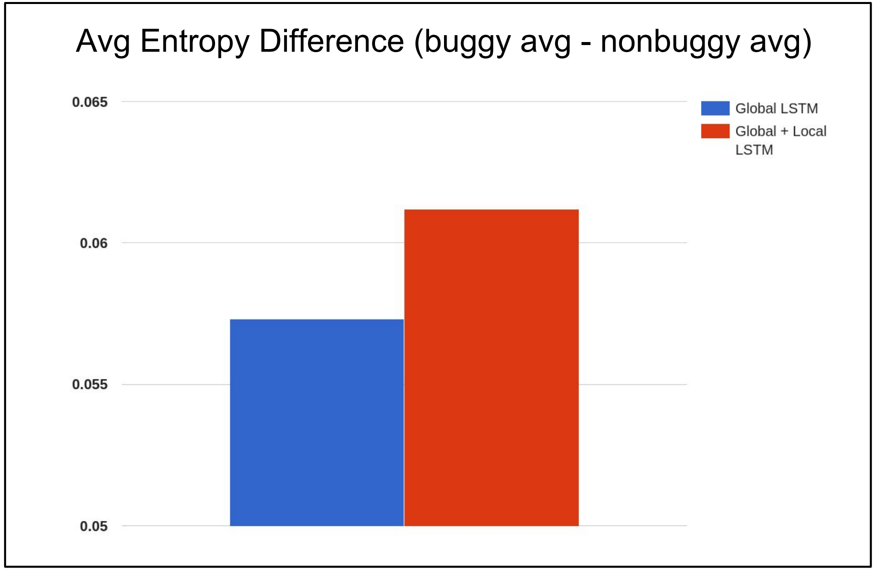
The “local LM” was trained on the remaining 9,416,098 lines

Dataset

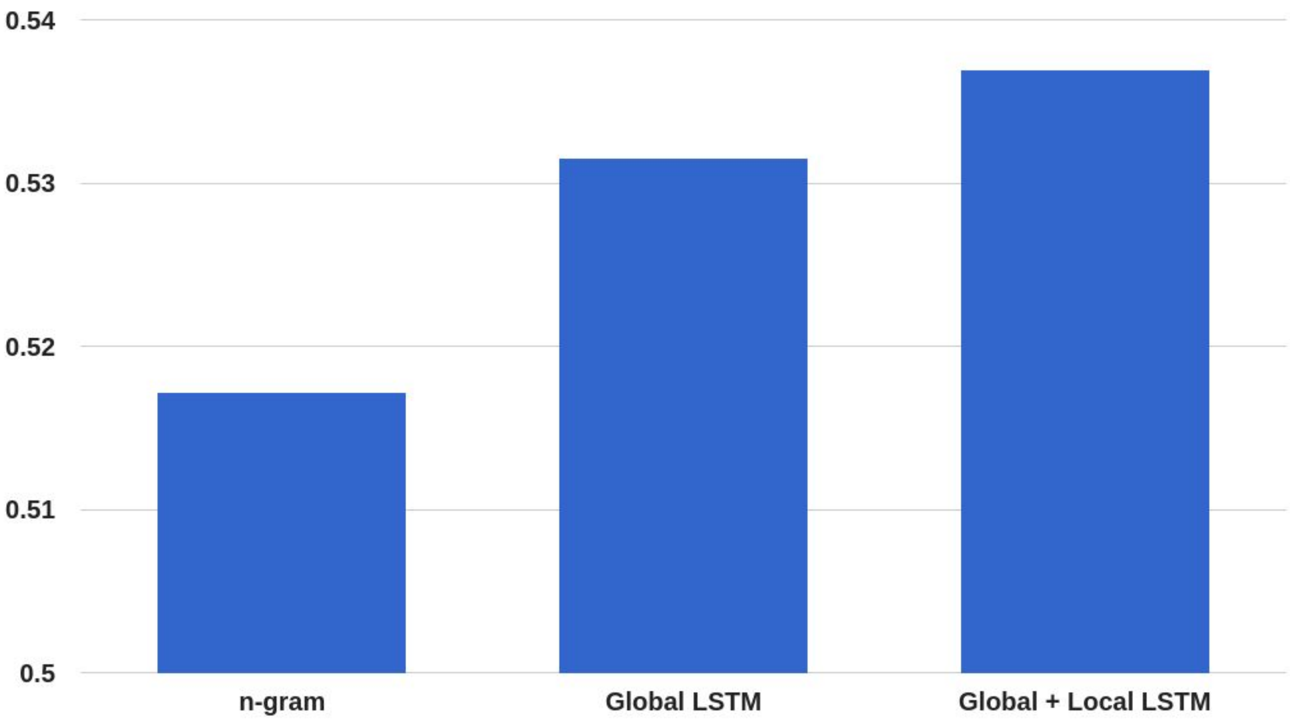


Average Entropy

	Buggy Lines		Non-Buggy Lines	
	Global LSTM	Global + Local LSTM	Global LSTM	Global + Local LSTM
Avg Entropy	1.6498	1.5842	1.5925	1.5230



AUC comparison



Buggy Language Model

Trained a “buggy language model” on the buggy lines (+/- 5 lines) on the training set.

Tested on the same lines as before

	Buggy Lines	**Non-Buggy Lines**
Avg Entropy	1.20028	1.18082

Future Work

1. Train the language model on many different projects of the same language (e.g. Java) in order to create a true model of the actual language.
2. Then, train a local language model on just the project of interest.