

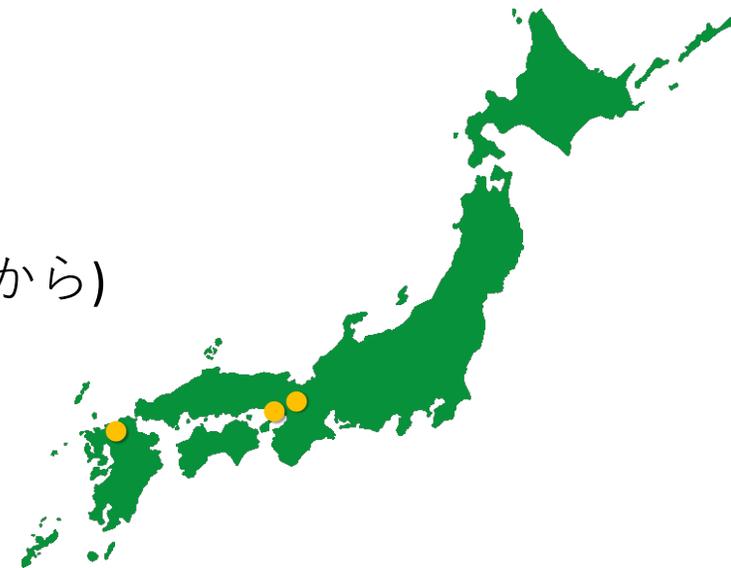
確率的言語モデルと ステガノグラフィ

京都大学
大学院情報学研究科
知能情報学コース
村脇 有吾



自己紹介: 村脇 有吾 (むらわき・ゆうご)

- 神戸市出身
- 京都大学大学院情報学研究科 准教授 (今日から)
- 専門は計算言語学・自然言語処理
 - 言葉を操れるコンピュータを作ります
 - 文系? 私はずっと理系です
 - 意外(?)と数学を使います



ここにあった
画像は削除さ
れました

突然ですがみなさんに質問です 

にすいという言葉を使いますか？

使うとしたらどんな意味で使いますか？

今日のお話

- 京都大学の紹介 (公開版では削除)
- 確率的言語モデル
- 言語ステガノグラフィ

今日のお話

- 京都大学の紹介
- 確率的言語モデル
- 言語ステガノグラフィ

Browser window showing the ChatGPT interface. The address bar displays `https://chat.openai.com/?model=text-d`. The page title is "ChatGPT".



Model selection buttons: **GPT-3.5** (active) and **GPT-4** (locked).

ChatGPT



Examples

"Explain quantum computing in simple terms" →

"Got any creative ideas for a 10 year old's birthday?" →



Capabilities

Remembers what user said earlier in the conversation

Allows user to provide follow-up corrections



Limitations

May occasionally generate incorrect information

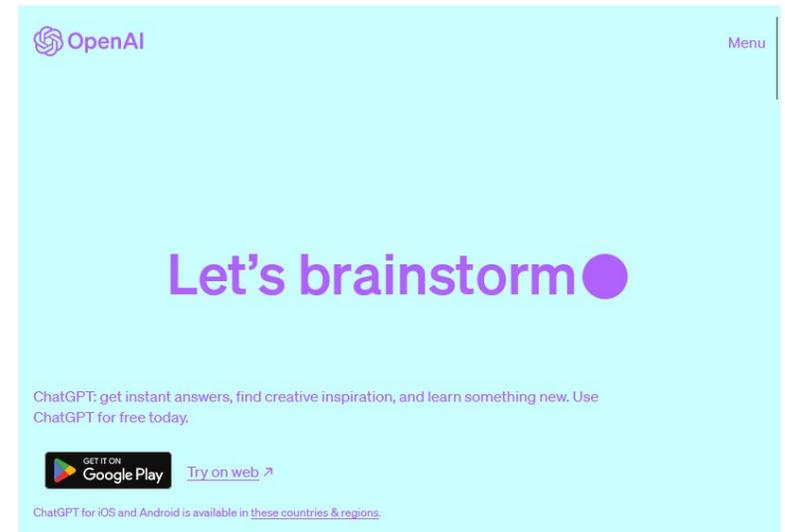
May occasionally produce harmful instructions or biased content

Send a message

Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts. [ChatGPT July 20 Version](#)

ChatGPTとは？

- OpenAIが2022年11月に公開した大規模**確率的言語モデル**に基づくチャットボット
- 日本語も流暢
- プログラムも書けます



ユーザ登録すれば無料で試せます

MU

三角形の面積を計算するプログラムを書いて。



三角形の面積を計算するためには、三角形の底辺と高さが必要です。以下はPythonで三角形の面積を計算するプログラム例です。

```
python Copy code  
  
def triangle_area(base, height):  
    area = 0.5 * base * height  
    return area  
  
# ユーザーから底辺と高さを入力してもらう場合  
try:  
    base = float(input("三角形の底辺の長さを入力してください: "))  
    height = float(input("三角形の高さを入力してください: "))  
  
    if base <= 0 or height <= 0:  
        print("底辺と高さは正の数である必要があります。")  
    else:  
        result = triangle_area(base, height)  
        print("三角形の面積は:", result)  
except ValueError:  
    print("無効な入力です。数値を入力してください。")
```

このプログラムでは、ユーザーから底辺と高さを入力してもらい、それを使って三角形の面積を計算します。底辺と高さは正の数である必要があります。入力が無効な場合（数値でない場合）はエラーメッセージが表示されず。

例えば、底辺が5.0、高さが8.0の場合、面積は20.0になります。

Regenerate

前半の目標

- ChatGPTのもとになった確率的言語モデルの基本原則を紹介します
- 高校レベルでは難しい話題もありますが、行けるところまで行きます！

特徴: テキストを前から順番に生成している

吾輩

特徴: テキストを前から順番に生成している

吾輩は猫である。名前は

特徴: テキストを前から順番に生成している

吾輩 は 猫 である 。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

特徴: テキストを前から順番に生成している

吾輩 は 猫 である 。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

- 単語 (word) を1個ずつ生成する関数 f を作ればよい

特徴: テキストを前から順番に生成している

吾輩 は 猫 である 。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

- 単語 (word) を1個ずつ生成する関数 f を作ればよい
- $w_k = f(w_1, \dots, w_{k-1})$
 - 入力: 既に生成した単語列 w_1, \dots, w_{k-1}
 - 出力: 次の単語 w_k

特徴: テキストを前から順番に生成している

吾輩 は 猫 である 。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

- 単語 (**w**ord) を1個ずつ生成する関数 f を作ればよい
- $w_k = f(w_1, \dots, w_{k-1})$
 - 入力: 既に生成した単語列 w_1, \dots, w_{k-1}
 - 出力: 次の単語 w_k
- f を繰り返し呼び出せばテキストが生成できる

特徴: テキストを前から順番に生成している

吾輩 は 猫 である 。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

特徴: テキストを前から順番に生成している

吾輩 は 猫 である。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

• 猫 = $f(\text{吾輩}, \text{は})$

特徴: テキストを前から順番に生成している

吾輩 は 猫 である。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

- 猫 = $f(\text{吾輩}, \text{は})$
- で = $f(\text{吾輩}, \text{は}, \text{猫})$

特徴: テキストを前から順番に生成している

吾輩 は 猫 である。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

- 猫 = $f(\text{吾輩}, \text{は})$
- で = $f(\text{吾輩}, \text{は}, \text{猫})$
- ある = $f(\text{吾輩}, \text{は}, \text{猫}, \text{で})$

特徴: テキストを前から順番に生成している

吾輩 は 猫 で ある 。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

- 猫 = $f(\text{吾輩}, \text{は})$
- で = $f(\text{吾輩}, \text{は}, \text{猫})$
- ある = $f(\text{吾輩}, \text{は}, \text{猫}, \text{で})$

※ f は可変長入力を受け付けます
(高校数学では出てきません)

終了条件 (どうやって生成を止める?)

- 特殊単語 `</s>` を用意しておく

終了条件 (どうやって生成を止める?)

- 特殊単語 `</s>` を用意しておく

吾輩 は 猫 で ある 。 `</s>`

w_1 w_2 w_3 w_4 w_5 w_6 w_7

終了条件 (どうやって生成を止める?)

- 特殊単語 $\langle /s \rangle$ を用意しておく

吾輩 は 猫 で ある 。 $\langle /s \rangle$

w_1 w_2 w_3 w_4 w_5 w_6 w_7

- $f(w_1, \dots, w_{k-1})$ が $w_k = \langle /s \rangle$ を返したら生成を止める

開始条件 (どうやって生成を始める?)

- 特殊単語 $\langle s \rangle$ を用意しておく

$\langle s \rangle$ 吾輩 は 猫 である 。 $\langle /s \rangle$

w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7

- 実は $w_k = f(w_0 = \langle s \rangle, w_1, \dots, w_{k-1})$ だが、簡単のため w_0 は省略

関数 f が満たすべき性質は？

吾輩 は 猫 である。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

関数 f が満たすべき性質は？

吾輩 は ~~猫~~^{ネズミ} である。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

- 「猫」を「ネズミ」に置き換えても自然なテキスト

関数 f が満たすべき性質は？

吾輩 は ~~猫~~^{ネズミ} である。 名前 は

w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

- 「猫」を「ネズミ」に置き換えても自然なテキスト
- $w_k = f(w_1, \dots, w_{k-1})$ となる w_k を直接1つに絞り込むのは筋が悪い

関数 f を2段階で構成

- 第1段階: 単語列 w_1, \dots, w_{k-1} に続き **そう** な単語の候補を挙げる

吾輩 は

w_1 w_2

猫
ネズミ
運動
⋮

関数 f を2段階で構成

- 第1段階: 単語列 w_1, \dots, w_{k-1} に続き **そう** な単語の候補を挙げる

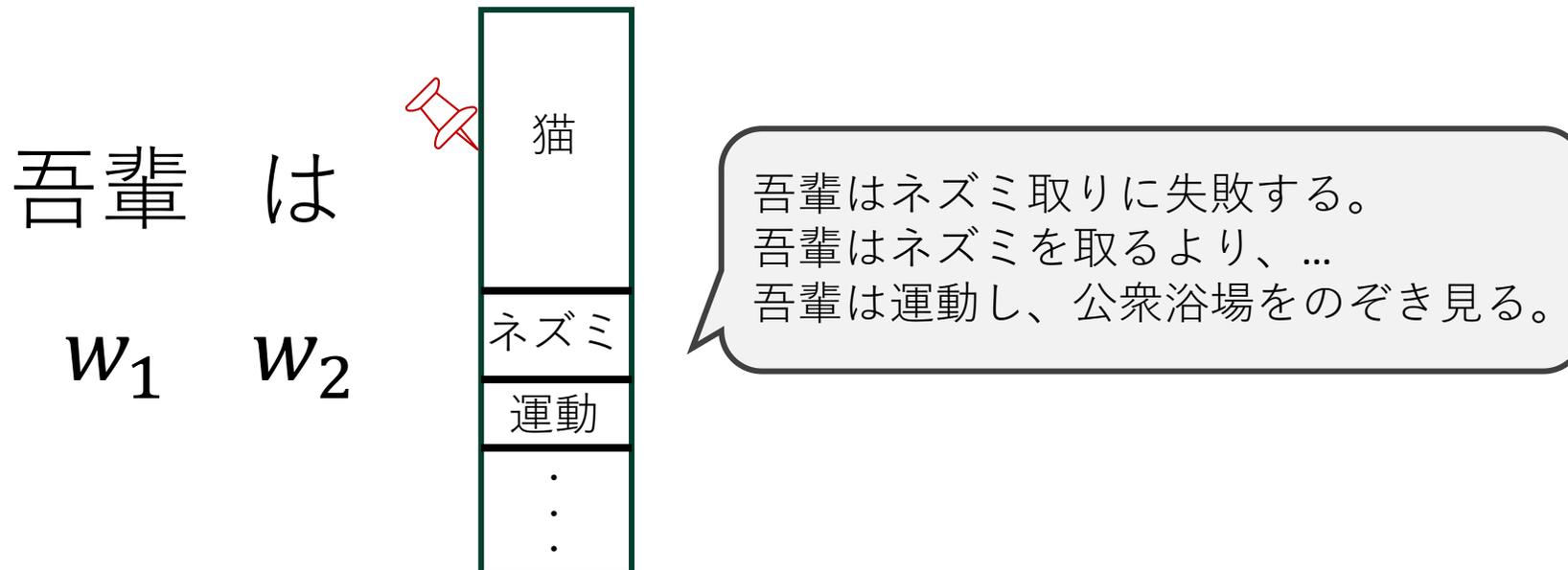
吾輩 は
 w_1 w_2



吾輩はネズミ取りに失敗する。
吾輩はネズミを取るより、...
吾輩は運動し、公衆浴場をのぞき見る。

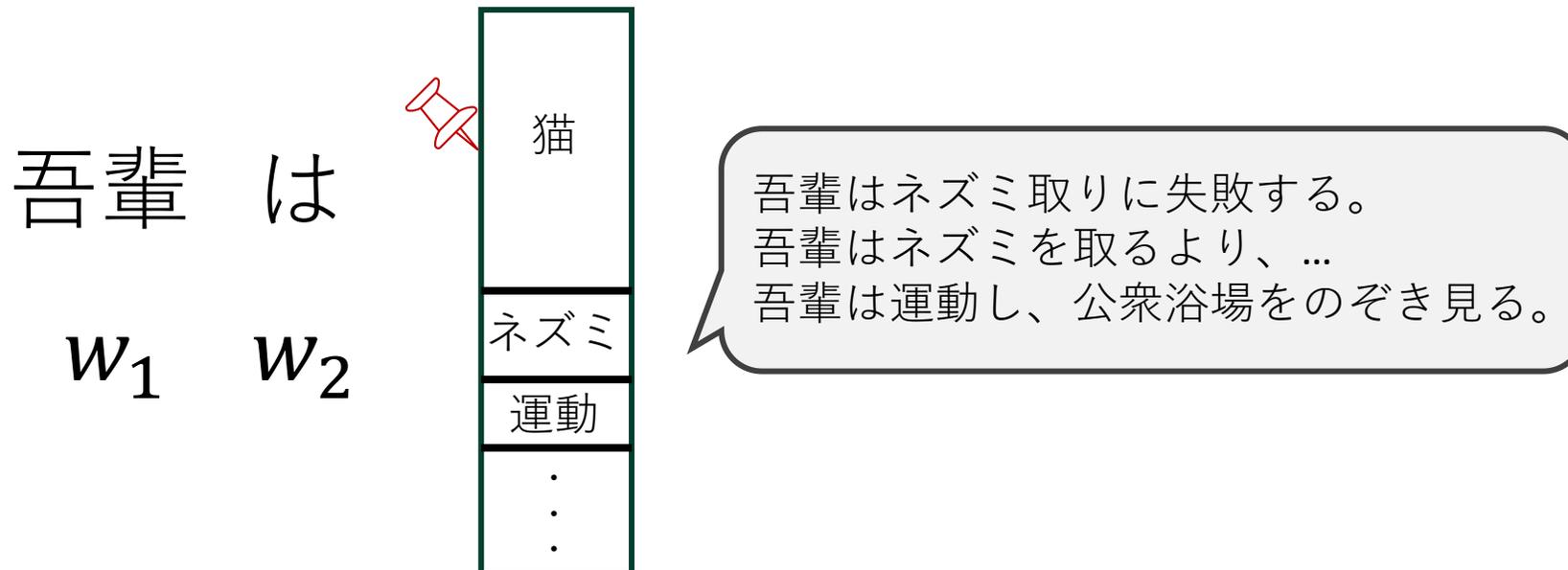
関数 f を2段階で構成

- 第1段階: 単語列 w_1, \dots, w_{k-1} に続き **そう** な単語の候補を挙げる
- 第2段階: 単語の候補から **ランダム** に単語を選択



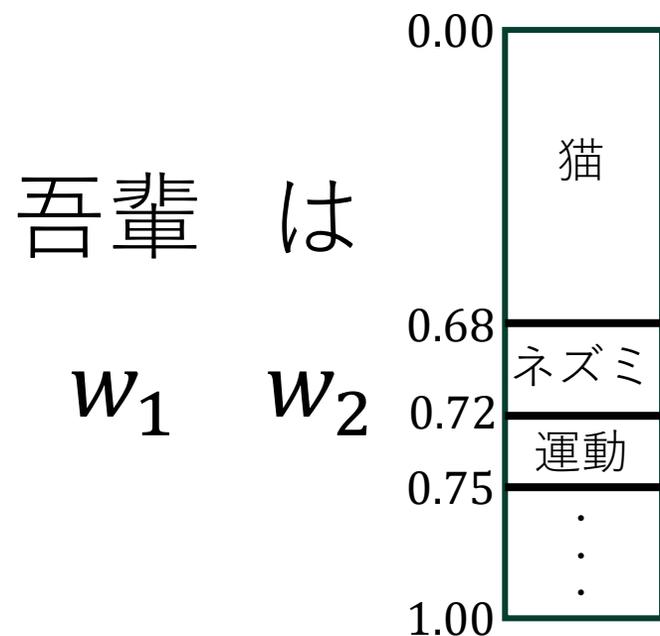
関数 f を2段階で構成

- 第1段階: 単語列 w_1, \dots, w_{k-1} に続き **そう** な単語の候補を挙げる
 - 第2段階: 単語の候補から **ランダム** に単語を選択
- ※ 数学の関数は同じ入力に対して常に同じ出力を返すので、
数学的な意味での関数ではなくなっています



第2段階:

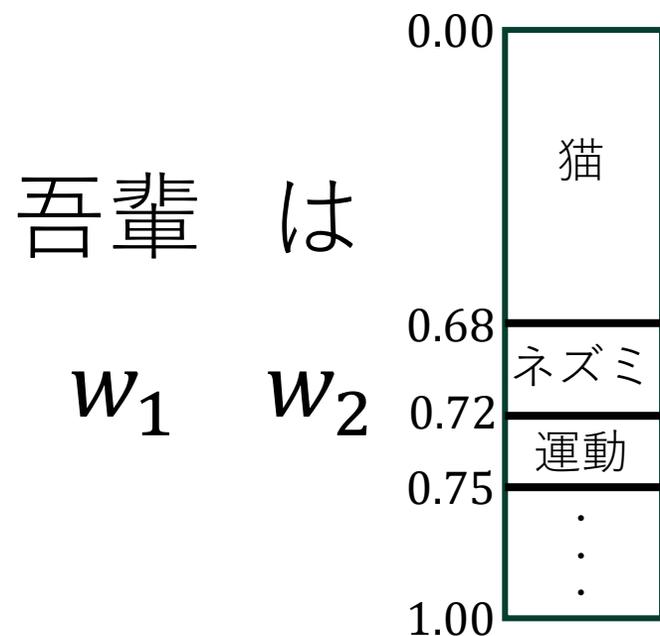
単語の候補からランダムに単語を選択



- 単語候補に確からしさに応じて正の値を割り振っておく
- 単語候補の割り振った値の総和は1としておく

第2段階:

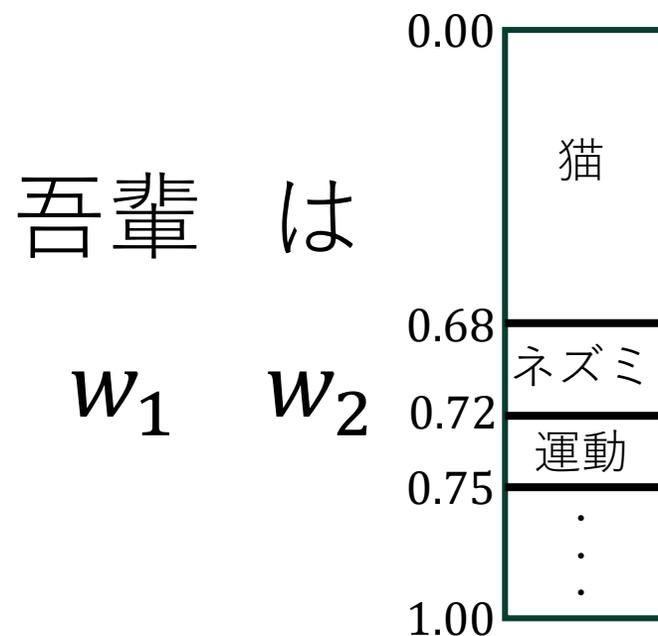
単語の候補からランダムに単語を選択



- 0から1までの範囲の擬似乱数 r を得る

第2段階:

単語の候補からランダムに単語を選択

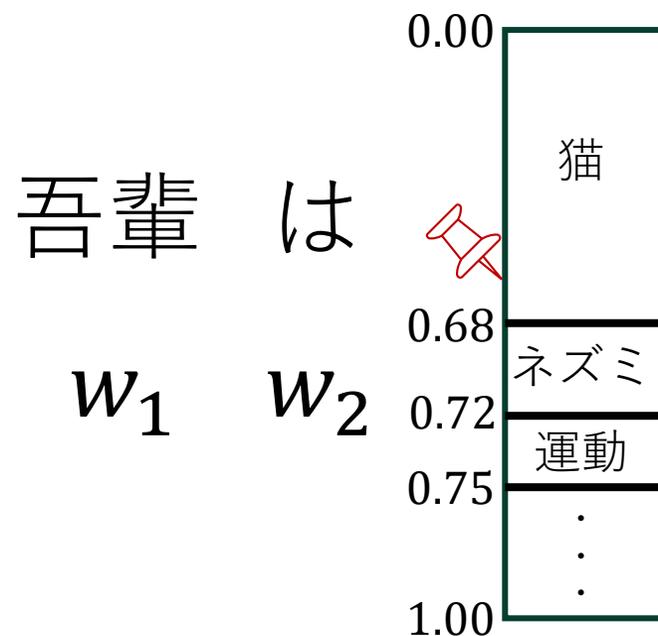


$$r = 0.5902$$

- 0から1までの範囲の擬似乱数 r を得る

第2段階:

単語の候補からランダムに単語を選択

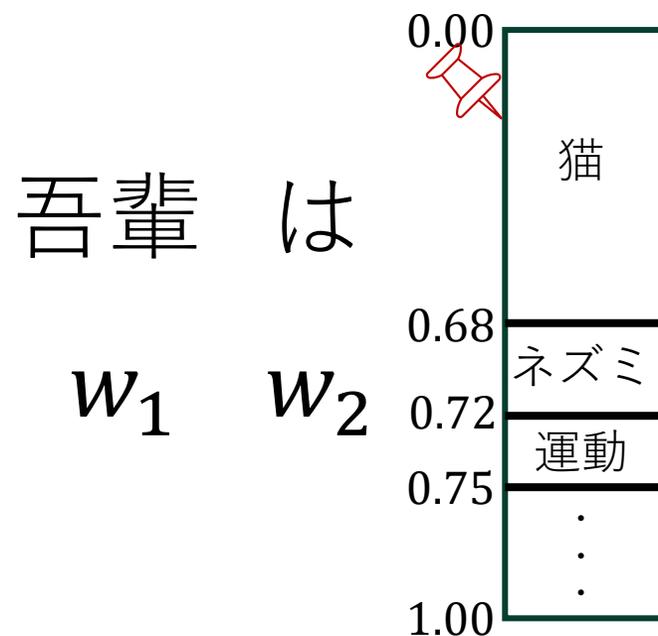


$$r = 0.5902$$

- 0から1までの範囲の擬似乱数 r を得る
- 擬似乱数の値に対応する単語を選択

第2段階:

単語の候補からランダムに単語を選択

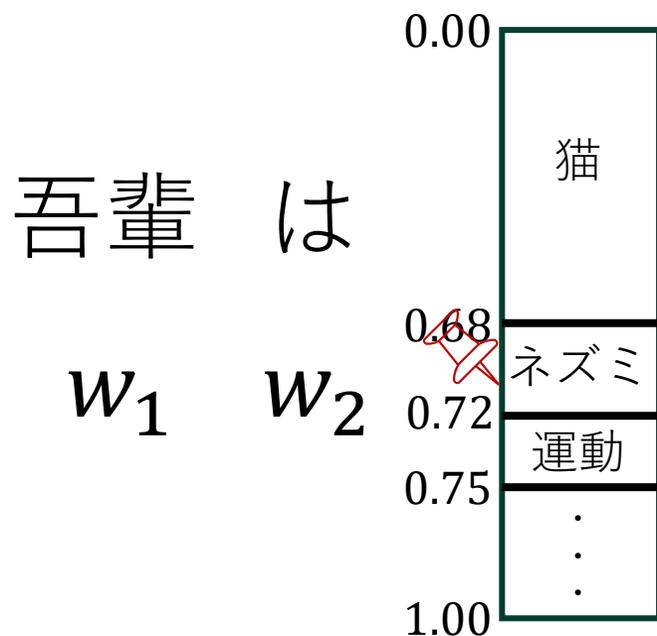


$$r = 0.2395$$

- 0から1までの範囲の擬似乱数 r を得る
- 擬似乱数の値に対応する単語を選択

第2段階:

単語の候補からランダムに単語を選択

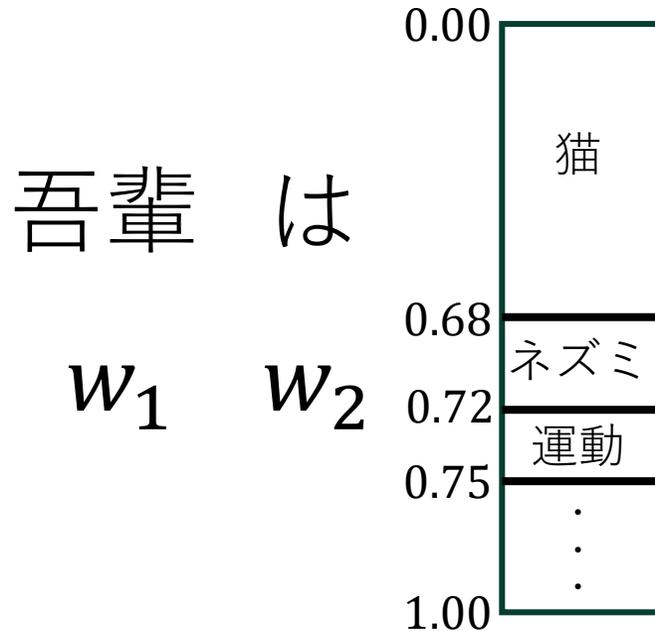


$$r = 0.7018$$

- 0から1までの範囲の擬似乱数 r を得る
- 擬似乱数の値に対応する単語を選択

第1段階:

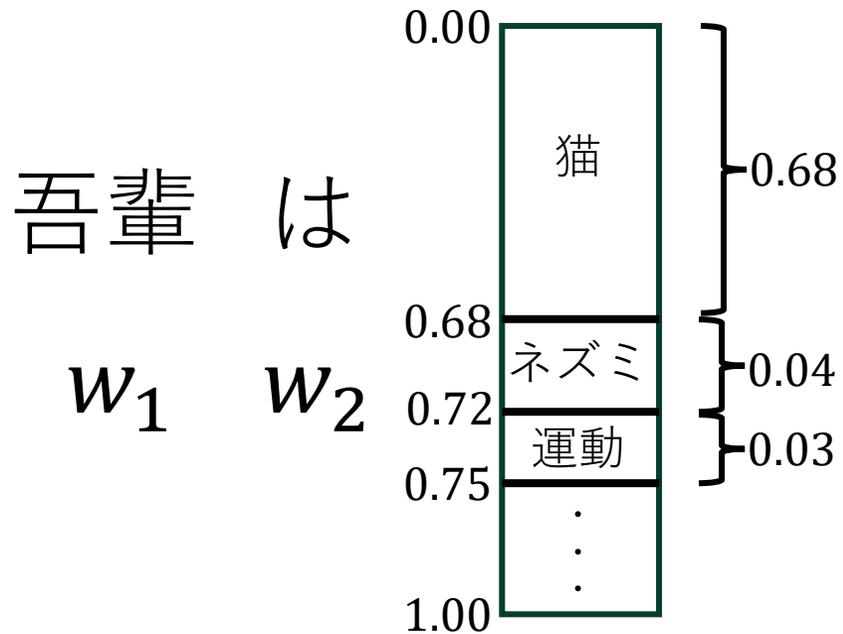
w_1, \dots, w_{k-1} に続きそうな単語候補を列挙



- 単語候補に確からしさに応じて正の値を割り振る
- 単語候補の割り振った値の総和は1とする

第1段階:

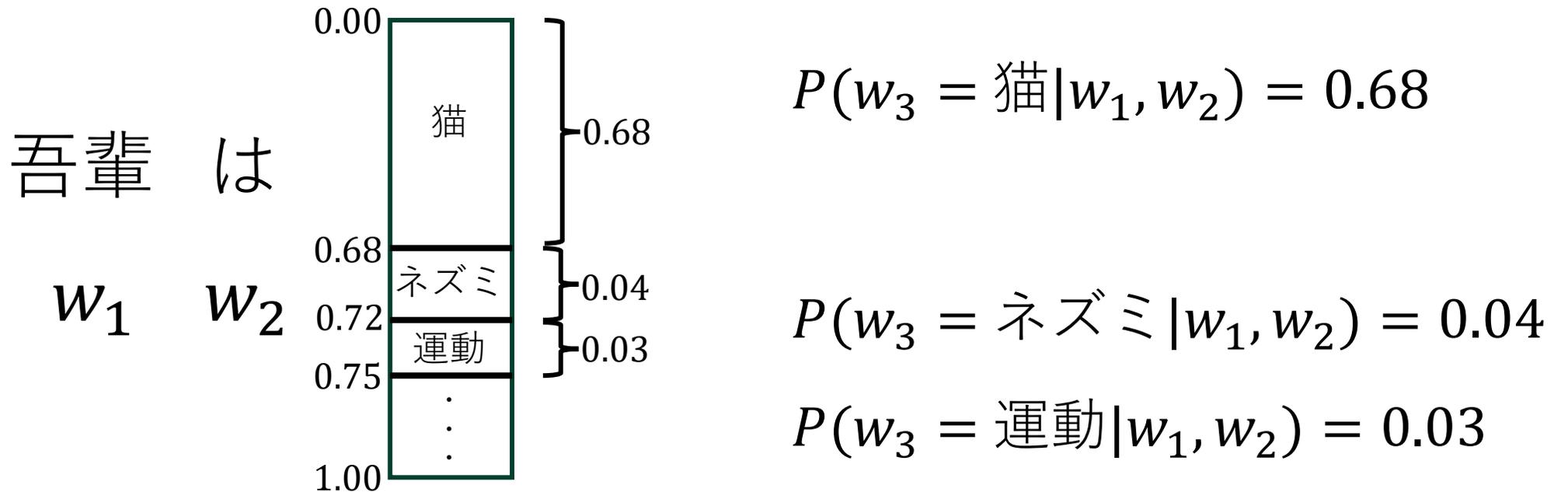
w_1, \dots, w_{k-1} に続きそのような単語候補を列挙



- 単語候補に確からしさに応じて正の値を割り振る
- 単語候補の割り振った値の総和は1とする

第1段階:

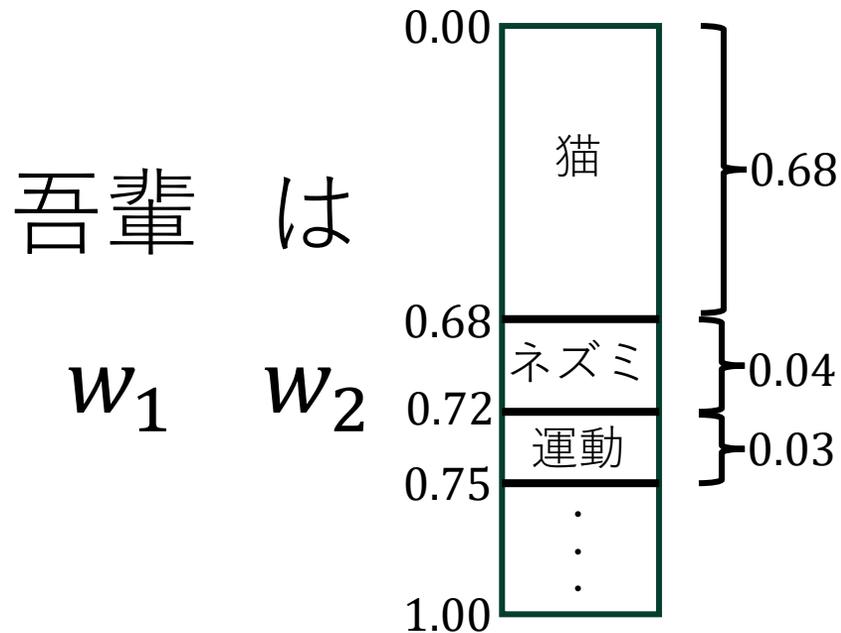
w_1, \dots, w_{k-1} に続きそうな単語候補を列挙



- 単語候補に確からしさに応じて正の値を割り振る
- 単語候補の割り振った値の総和は1とする

第1段階:

w_1, \dots, w_{k-1} に続きそうな単語候補を列挙



$$P(w_3 = \text{猫} | w_1, w_2) = 0.68$$

w_1, w_2 が与えられたもとで
 $w_3 = \text{猫}$ である確率

$$P(w_3 = \text{ネズミ} | w_1, w_2) = 0.04$$

$$P(w_3 = \text{運動} | w_1, w_2) = 0.03$$

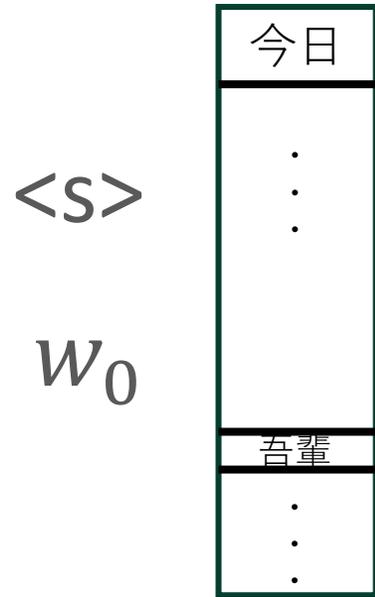
- 単語候補に確からしさに応じて正の値を割り振る
- 単語候補の割り振った値の総和は1とする

まとめ: 関数 f を使ったテキスト生成

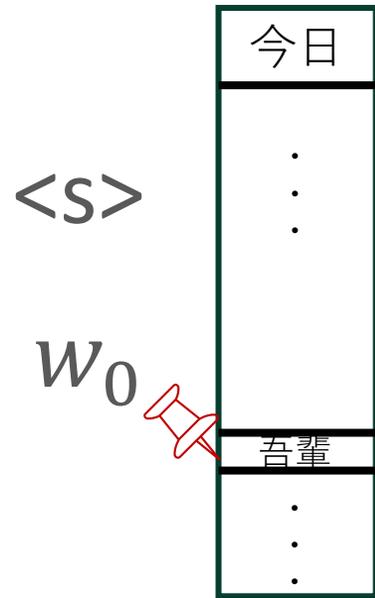
$\langle s \rangle$

w_0

まとめ: 関数 f を使ったテキスト生成



まとめ: 関数 f を使ったテキスト生成

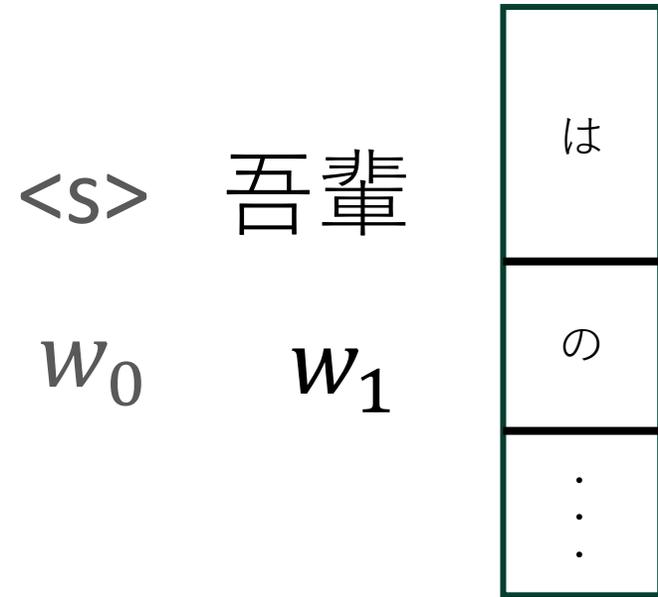


まとめ: 関数 f を使ったテキスト生成

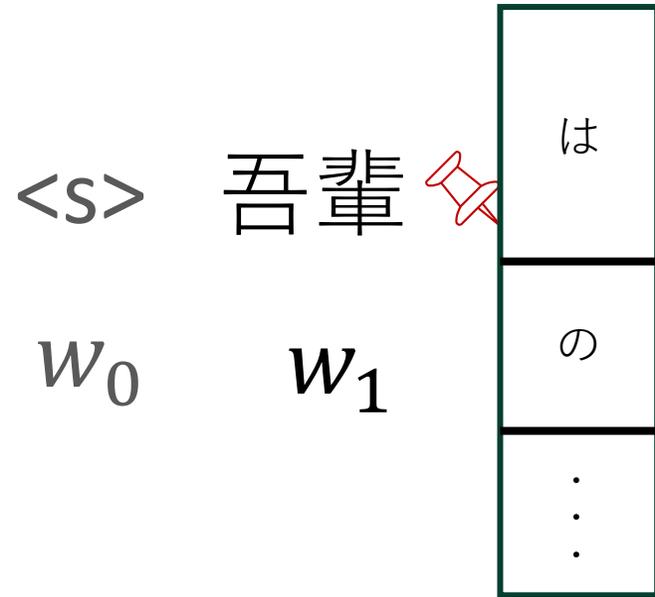
<S> 吾輩

w_0 w_1

まとめ: 関数 f を使ったテキスト生成



まとめ: 関数 f を使ったテキスト生成



まとめ: 関数 f を使ったテキスト生成

<S> 吾輩 は

w_0 w_1 w_2

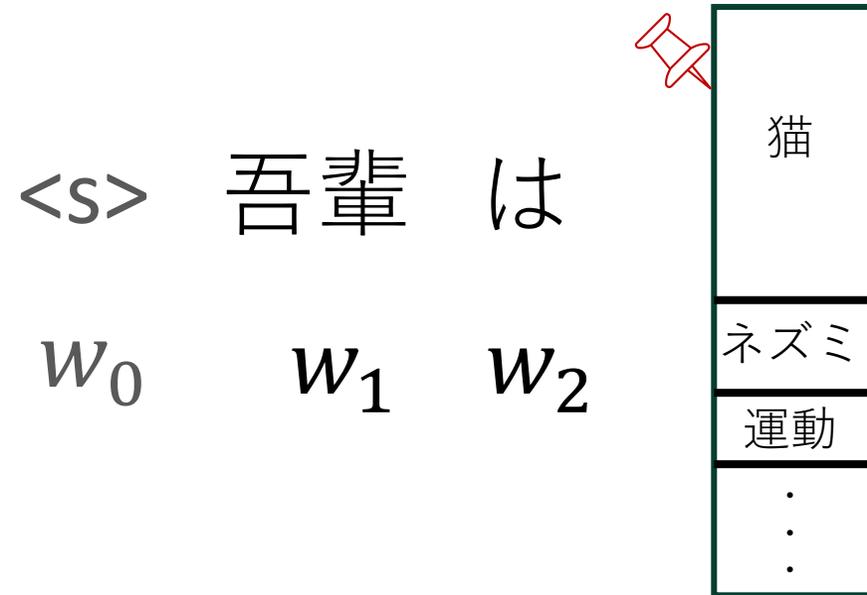
まとめ: 関数 f を使ったテキスト生成

<S> 吾輩 は

w_0 w_1 w_2



まとめ: 関数 f を使ったテキスト生成



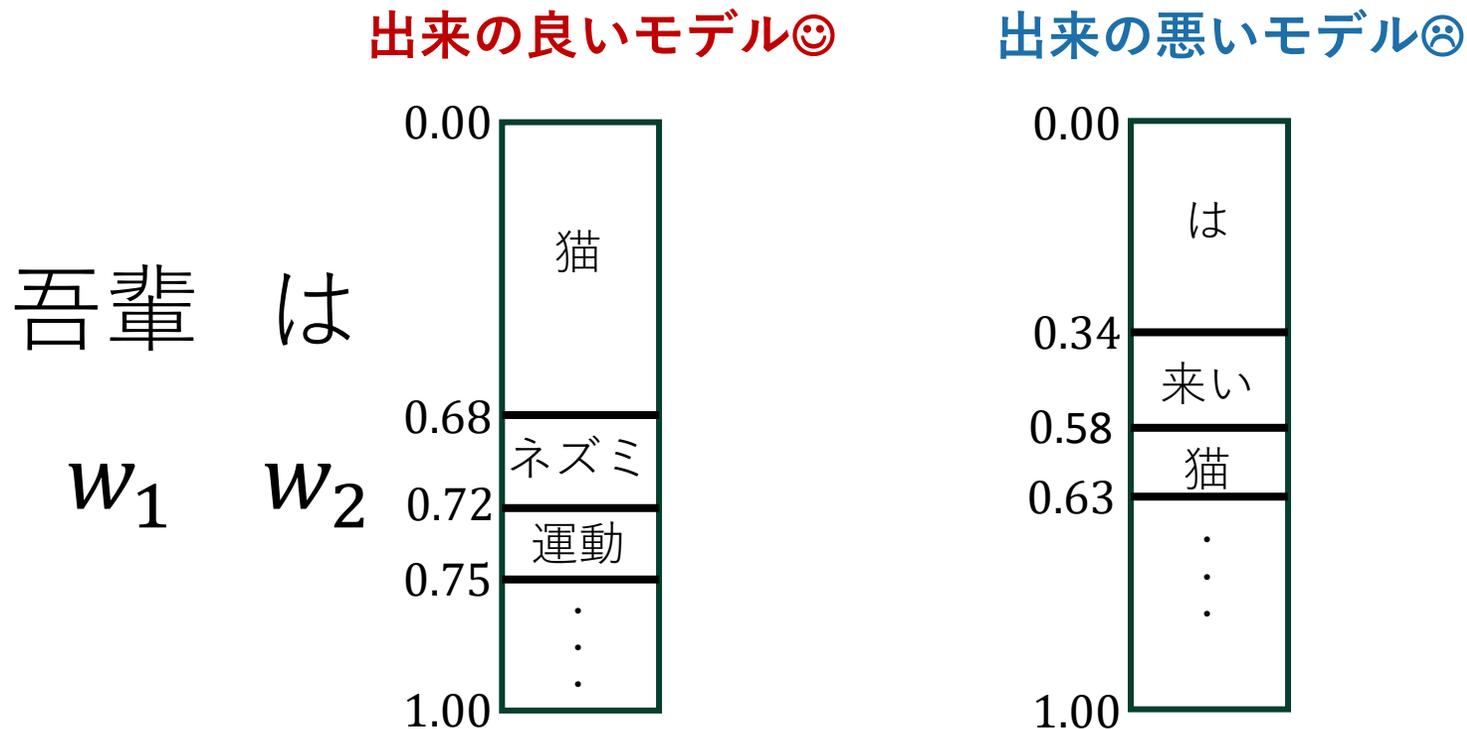
まとめ: 関数 f を使ったテキスト生成

<s> 吾輩 は 猫
 w_0 w_1 w_2 w_3

</s> が出るまで
これを繰り返す

次の課題: モデルを作って確率を推定

- 確率 $P(w_k | w_1, \dots, w_{k-1})$ の真の値は神様しか知らない
- 良い値を返すような**モデル** (確率的言語モデル) を作る



確率的言語モデルの作り方 (簡易版)

大規模テキスト
(例: 3,000億単語)

吾輩は猫である。...

「吾輩は猫である」を読んだ。

吾輩はネズミを取るより、...

そして吾輩は運動し、...

確率的言語モデルの作り方 (簡易版)

大規模テキスト
(例: 3,000億単語)

吾輩は猫である。...

「吾輩は猫である」を読んだ。

吾輩はネズミを取るより、...

そして吾輩は運動し、...

$$\begin{aligned} \bullet P(\text{猫} | \text{吾輩, は}) &= \frac{\text{頻度}(\text{吾輩, は, 猫})}{\text{頻度}(\text{吾輩, は})} \\ &= \frac{13100}{19265} = 0.68 \end{aligned}$$

確率的言語モデルの作り方 (簡易版)

大規模テキスト
(例: 3,000億単語)

吾輩は猫である。...

「吾輩は猫である」を読んだ。

吾輩はネズミを取るより、...

そして吾輩は運動し、...

$$\begin{aligned} \bullet P(\text{猫} | \text{吾輩, は}) &= \frac{\text{頻度}(\text{吾輩, は, 猫})}{\text{頻度}(\text{吾輩, は})} \\ &= \frac{13100}{19265} = 0.68 \end{aligned}$$

$$\begin{aligned} \bullet P(\text{ネズミ} | \text{吾輩, は}) &= \frac{\text{頻度}(\text{吾輩, は, ネズミ})}{\text{頻度}(\text{吾輩, は})} \\ &= \frac{770}{19265} = 0.04 \end{aligned}$$

簡易版の名は N グラム言語モデル

- w_1, \dots, w_{k-1} すべての単語列を考慮するのは大変なので、直前の $N - 1$ 単語だけを使って次の単語の確率を推定

$$P(w_k | w_1, w_2, \dots, w_{k-1}) \approx P(w_k | w_{k-N+1}, \dots, w_{k-1})$$

- 例えば3グラムモデルは直前2単語を考慮

- $P(\text{運動} | \text{そして, 吾輩, は}) \approx P(\text{運動} | \text{吾輩, は}) = \frac{\text{頻度}(\text{吾輩, は, 運動})}{\text{頻度}(\text{吾輩, は})}$

簡易版の名は N グラム言語モデル

- w_1, \dots, w_{k-1} すべての単語列を考慮するのは大変なので、直前の $N - 1$ 単語だけを使って次の単語の確率を推定

$$P(w_k | w_1, w_2, \dots, w_{k-1}) \approx P(w_k | \overline{w_{k-N+1}, \dots, w_{k-1}})$$

近似

- 例えば3グラムモデルは直前2単語を考慮

- $P(\text{運動} | \text{そして, 吾輩, は}) \approx P(\text{運動} | \text{吾輩, は}) = \frac{\text{頻度}(\text{吾輩, は, 運動})}{\text{頻度}(\text{吾輩, は})}$

Nグラム言語モデルからの生成例

「レンタ・カーは空のグラスを手にとり、蛇腹はすっかり暗くなっていた。それはまるで獲物を咀嚼しているようだった。彼は僕と同じようなものですね」と私は言った。「でもあなたはよく女の子に爪切りを買った。そしてその何かを振り払おうとしたが、今では誰にもできやしないのよ。私は長靴を棚の上を乗り越えるようにした。...

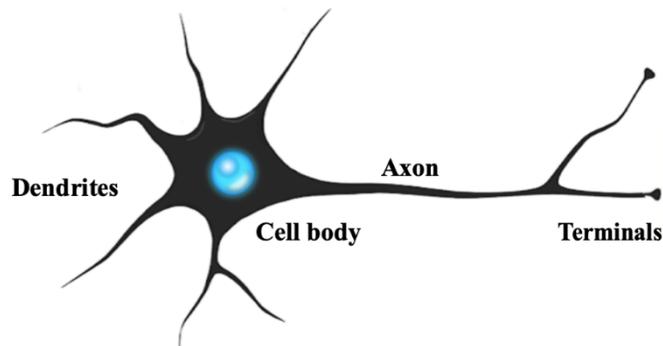
Nグラム言語モデルからの生成例

「レンタ・カーは空のグラスを手にとり、蛇腹はすっかり暗くなっていた。それはまるで獲物を咀嚼しているようだった。彼は僕と同じようなものですね」と私は言った。「でもあなたはよく女の子に爪切りを買った。そしてその何かを振り払おうとしたが、今では誰にもできやしないのよ。私は長靴を棚の上を乗り越えるようにした。...

局所的には自然なつながりに見えるけど、
全体として支離滅裂...

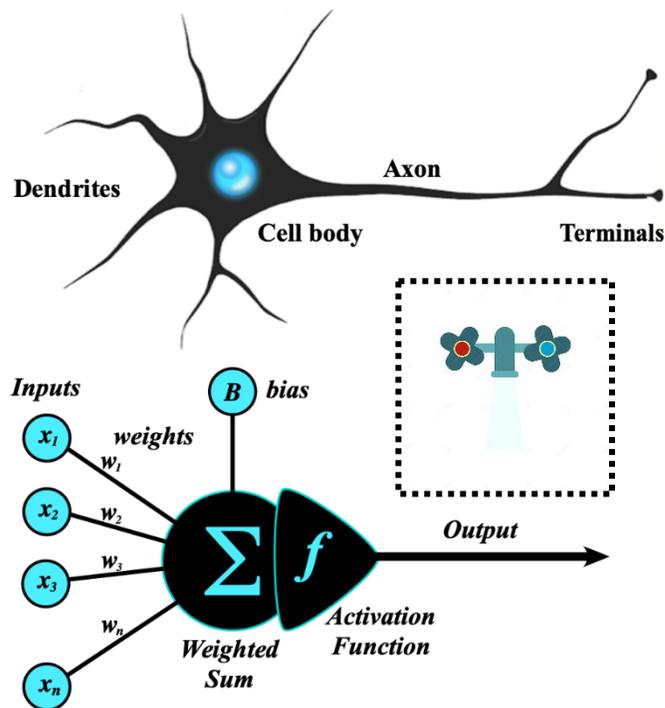
最近の強力な確率的言語モデルは (人工)ニューラルネットで構成されています

※ ここからは厳密な説明は諦めて、雰囲気だけ紹介していきます



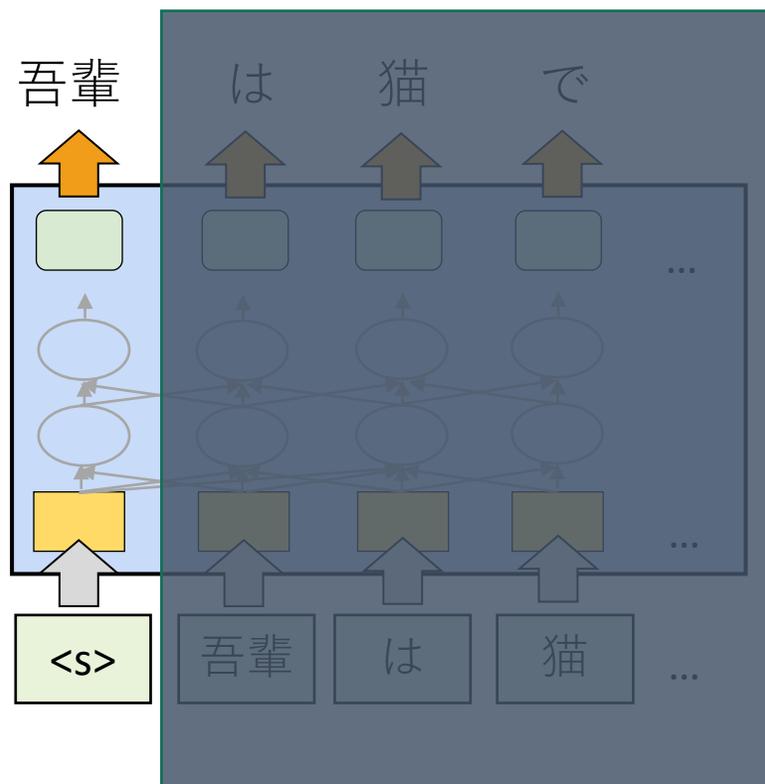
- 人間の脳は約**1000**億個のニューロンからなるといわれています
- ニューロン同士は互いにつながってネットワークを構成しています
- 隣のニューロンから来る電気刺激を受けて、電気刺激を出力します
- **1個1個**は単純な要素ですが、全体として高度な処理ができています

最近の強力な確率的言語モデルは (人工)ニューラルネットで構成されています

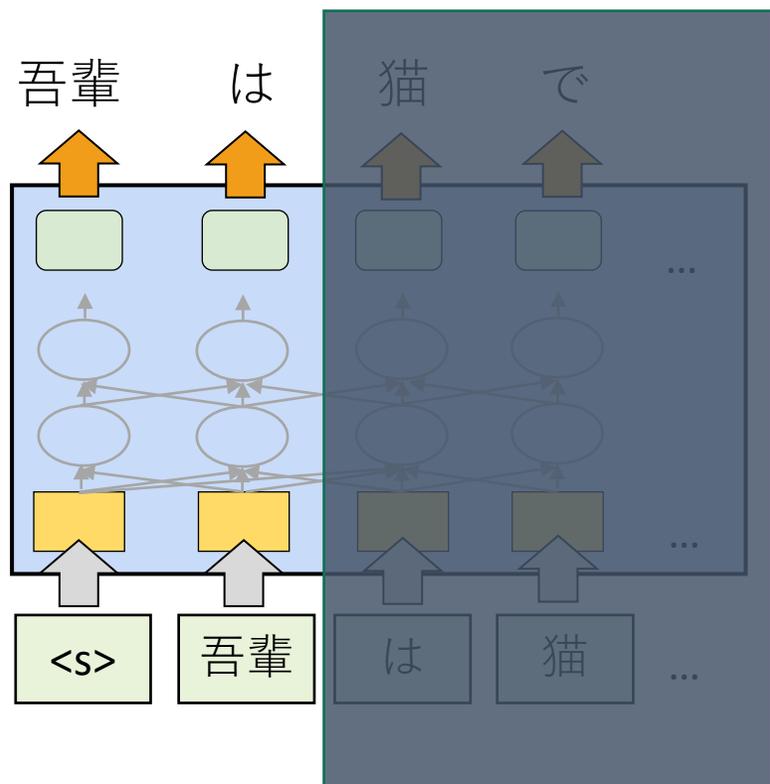


- 人工ニューロンは生体ニューロンを模して作られています
- 前のニューロンから送られた数値を加工して次のニューロンに数値を送ります
- 数値を加工するためのパラメータは水道のつまみにたとえられます
- ネットワークを使って数値を段階的に加工することで、望みの最終出力を得ます

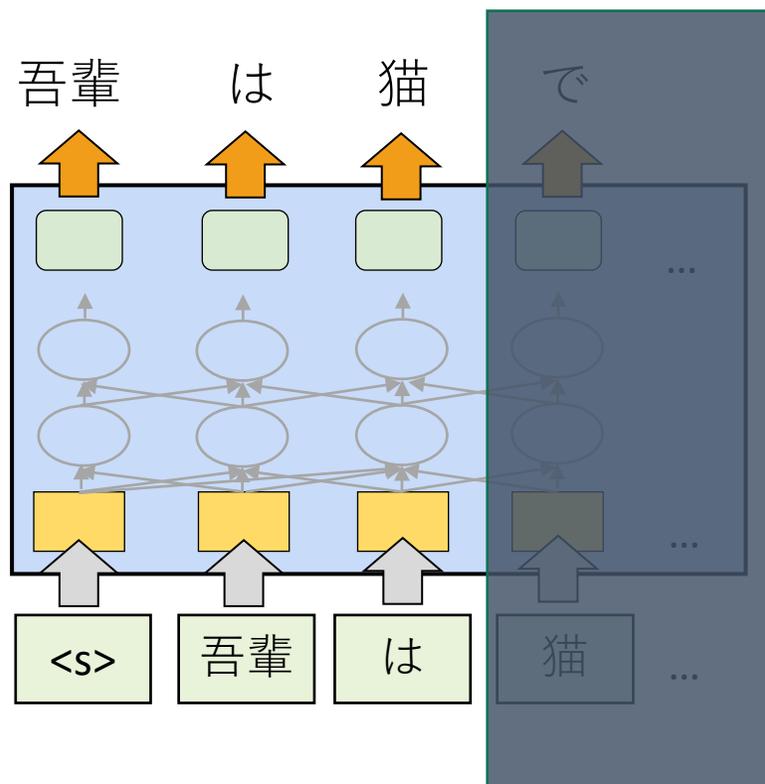
ニューラル言語モデルからの生成



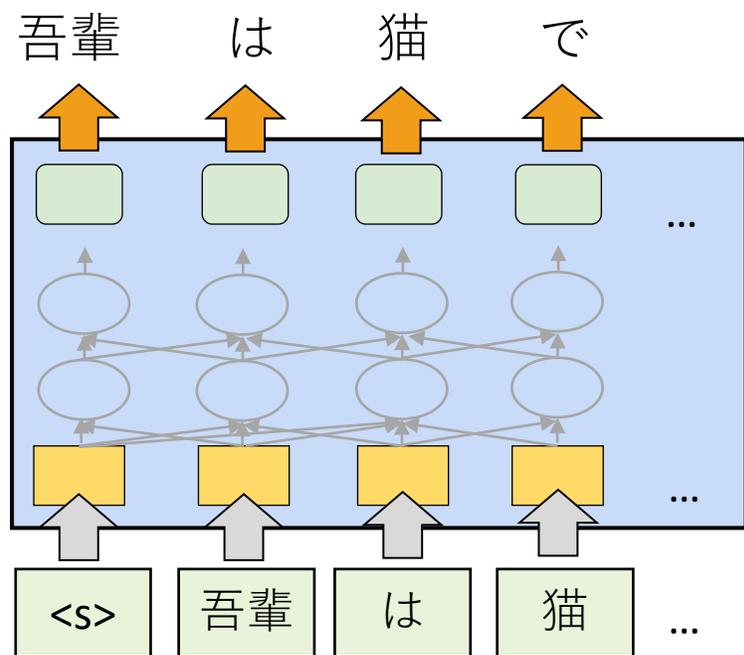
ニューラル言語モデルからの生成



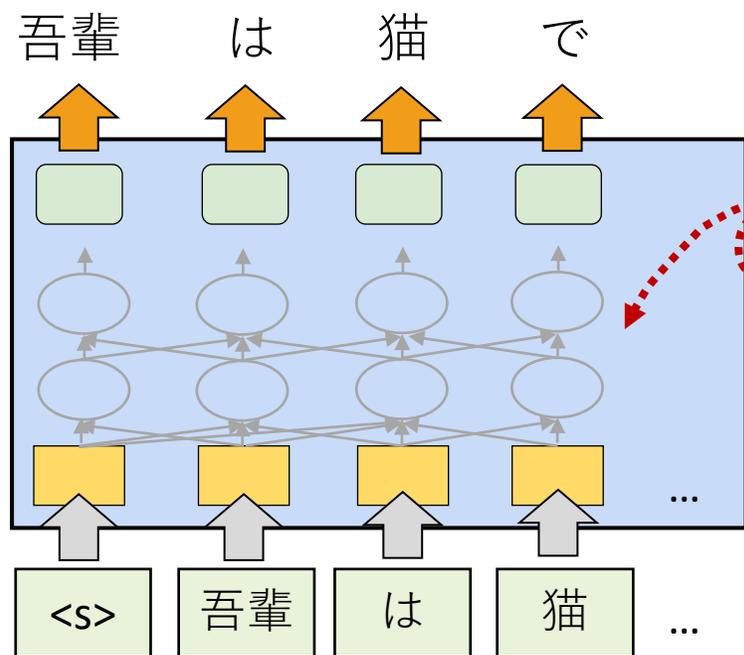
ニューラル言語モデルからの生成



ニューラル言語モデルからの生成

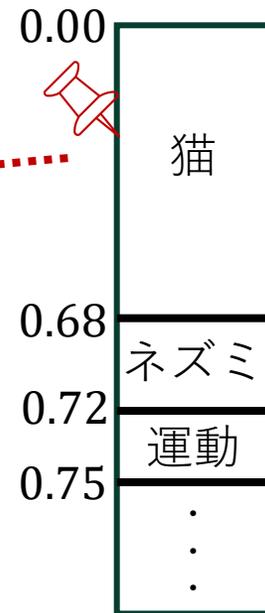
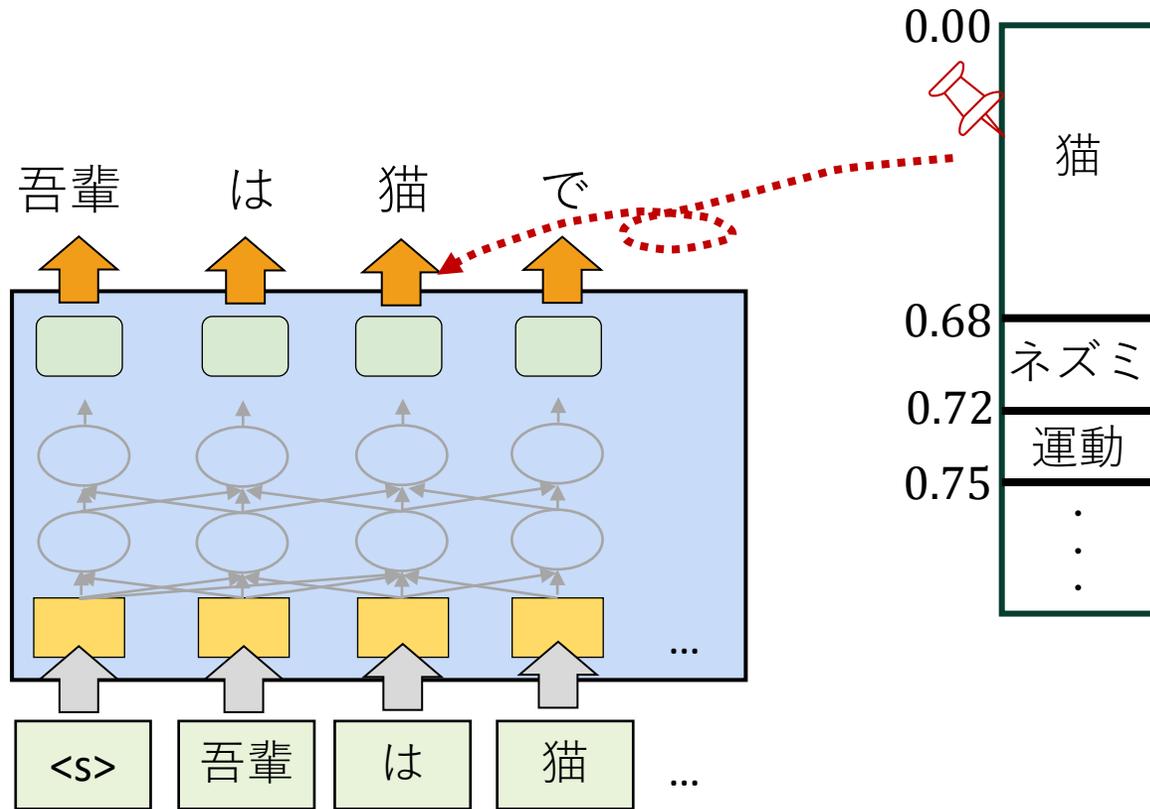


ニューラル言語モデルの図解 1/4



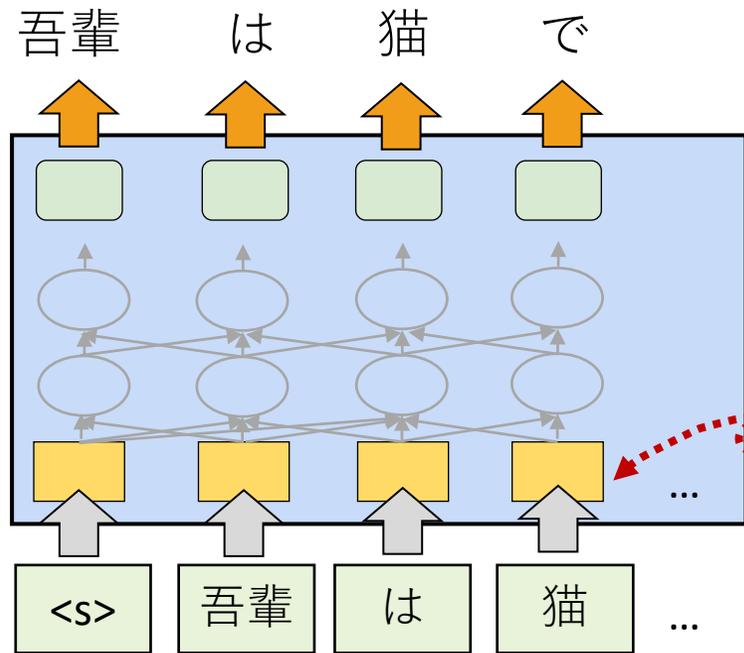
- 入力をパラメータ (≡つまみ🔩) を使って段階的に加工しています
- パラメータの数は数億から数千億個と膨大です

ニューラル言語モデルの図解 2/4



- 出力として単語候補の確率 $P(w_k | w_1, \dots, w_{k-1})$ を出力しています
- ランダムに次の単語を選ぶのはこれまでと同じです

ニューラル言語モデルの図解 3/4



- 単語はニューラルネットの入口で数値の羅列に変換します

<s> [-0.034, 0.002, -0.692, ...]

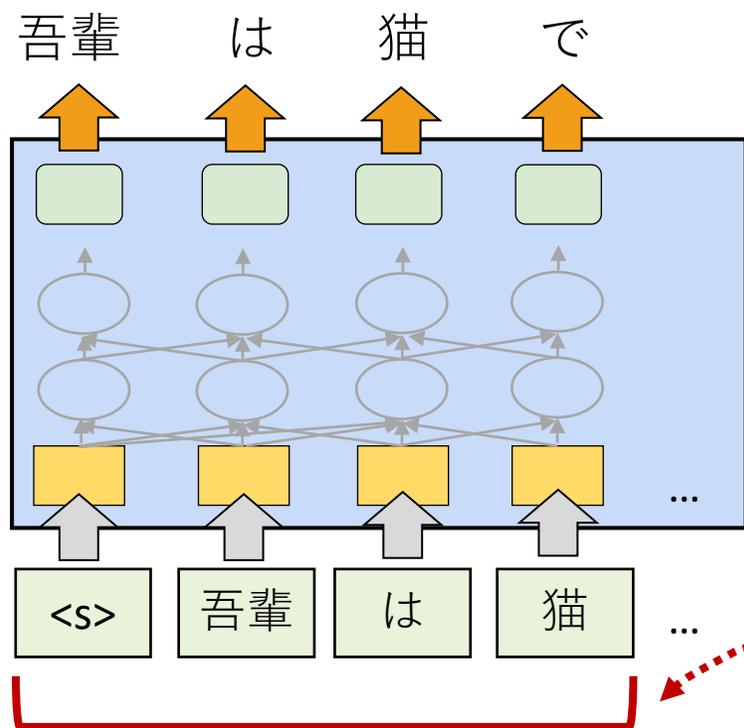
吾輩 [-0.022, -0.016, 0.029, ...]

は [-0.004, 0.153, 0.092, ...]

猫 [-0.037 - 0.012, 0.024, ...]

- これらの数値もパラメータです

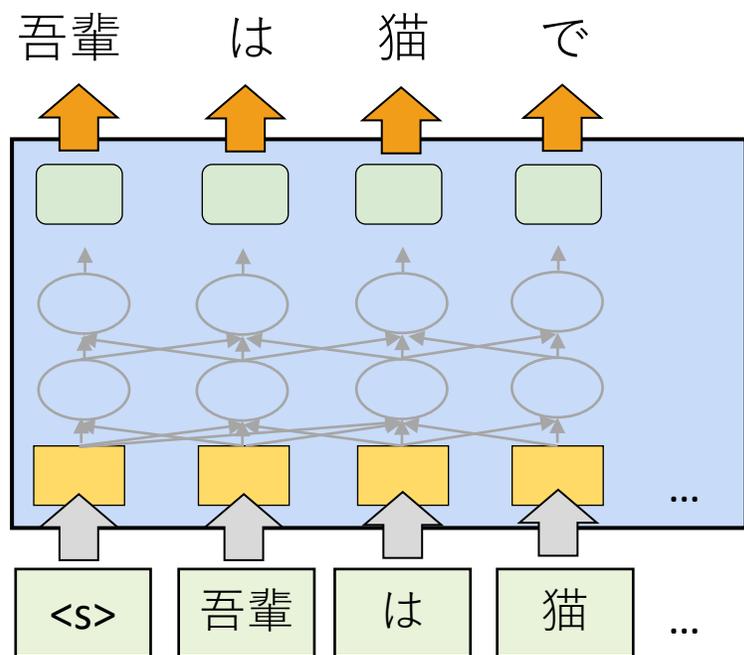
ニューラル言語モデルの図解 4/4



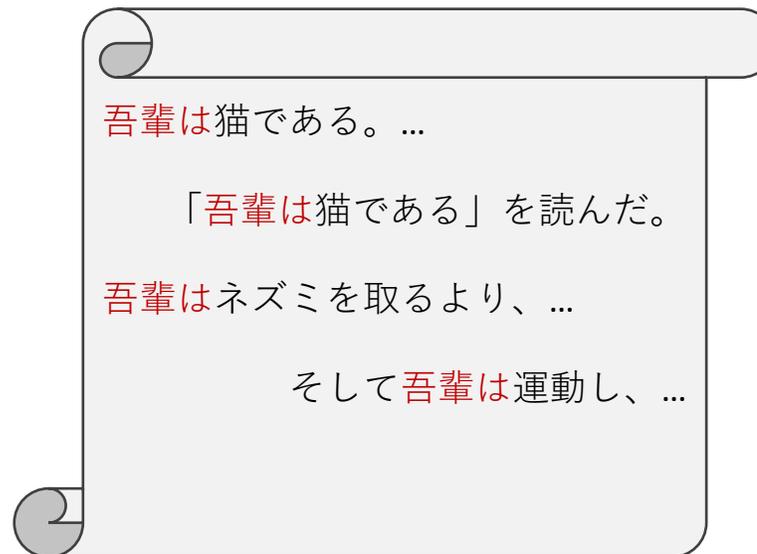
N グラム言語モデルと違って
すべての単語列を利用します

ニューラル言語モデルのパラメータ推定

大規模テキスト
(例: 3,000億単語)

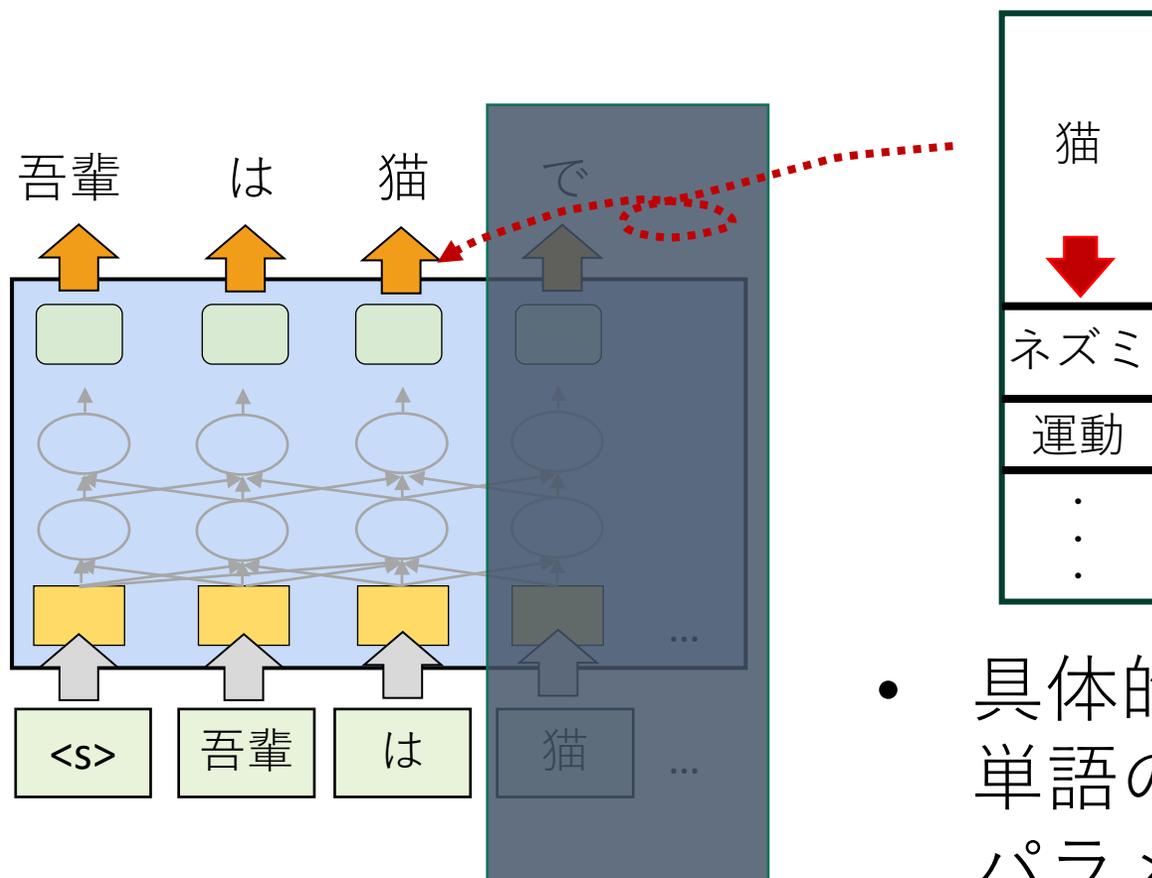


+



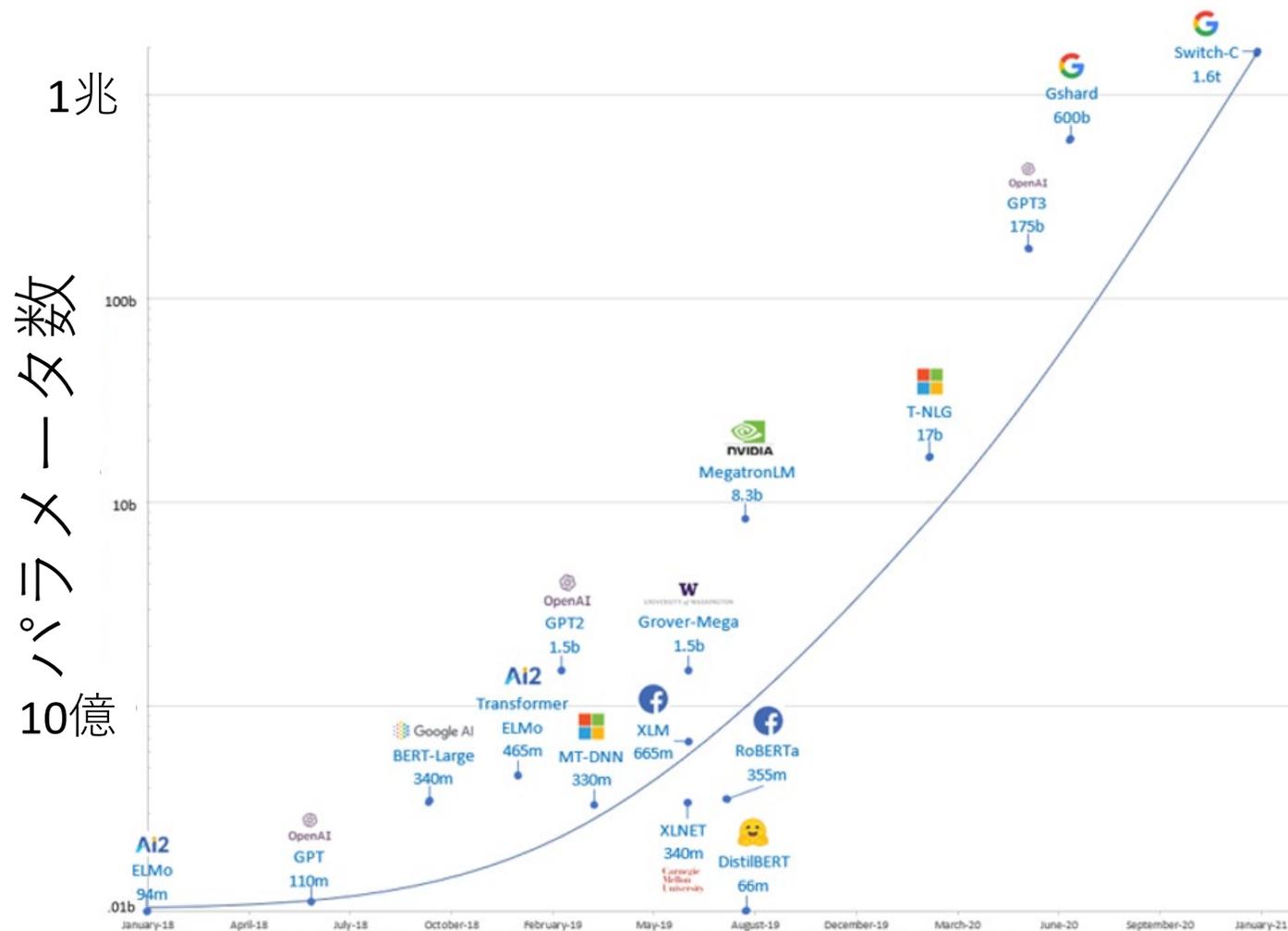
大規模テキストを使って入力に対して適切な出力が得られるようにパラメータ (≡つまみ🔧) を調整します

ニューラル言語モデルのパラメータ推定



- 具体的には猫の確率を上げて、他の単語の確率を下げるようにパラメータ (≡つまみ🔧) を調整します
- 実はここで微分が大活躍します

パラメータを増やすと性能が上がることがわかっており、大規模化競争が進んでいます



なぜ大規模化すると賢くなる？



- 研究途上でまだよくわかっていません...
- 大規模テキストには人間が持つ知識がかなり網羅的に書いてあり、大きなモデルほどそれらを覚えやすいようです
- 単語列を丸覚えするのではなく、一度数値列に変換して加工することにより、類推が働くようになるようです

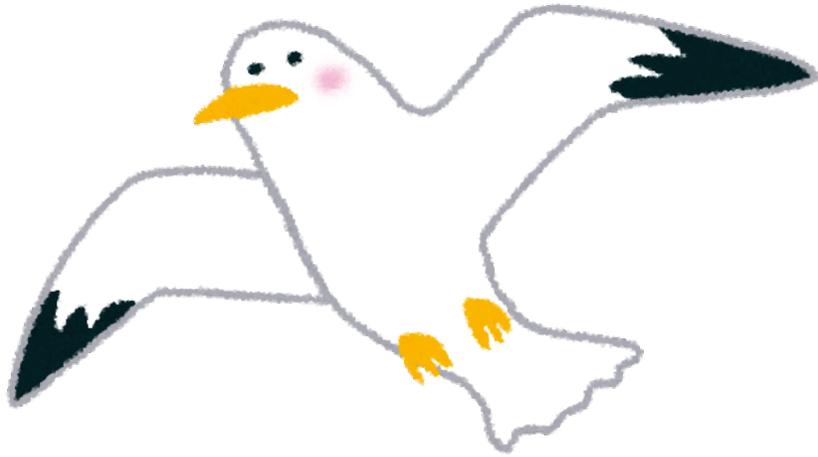
確率的言語モデル: まとめ

- テキストを生成するには**1**単語ずつ出力すればよい
- 関数 $f(w_1, \dots, w_{k-1})$ を作って、特殊単語 $\langle /s \rangle$ を出力するまで繰り返し f を呼び出せばよい
- f は2段階で構成します
 - 第1段階: 単語列 w_1, \dots, w_{k-1} に続く単語候補に確率を割り振る
 - 第2段階: それらの確率に従ってランダムに単語を選択
- 単語の確率の推定には最近ではニューラル言語モデルを使います

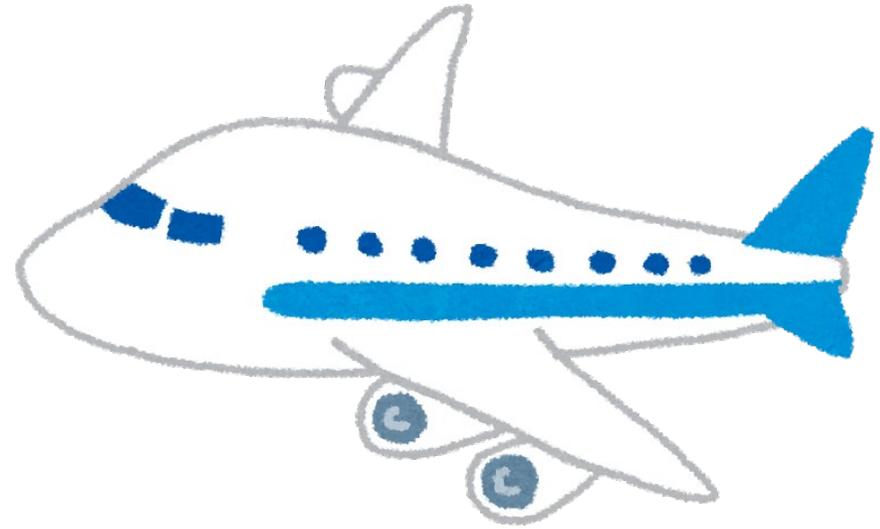
今日のお話

- 京都大学の紹介
- 確率的言語モデル
- 言語ステガノグラフィ

ニューラル言語モデルは
人のように言語を操っている？

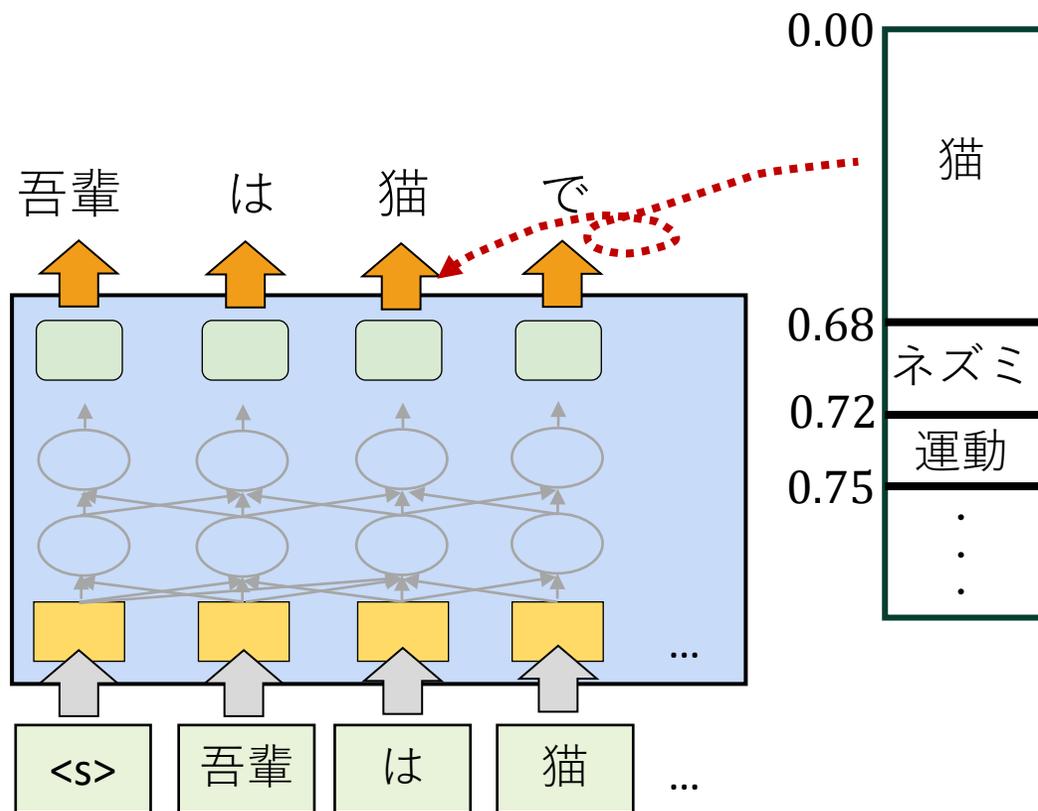


VS

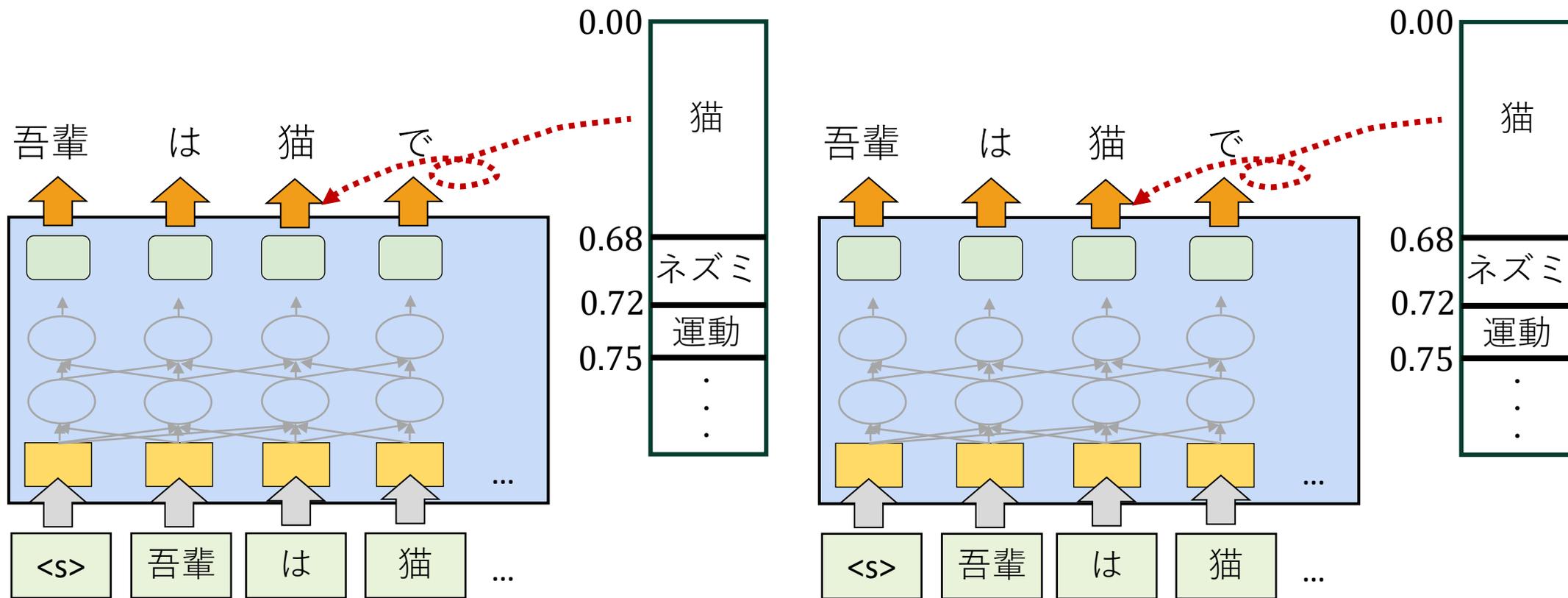


- 鳥が飛ぶ (:: 人が話す) のをまねるつもりだったのに、飛行機 (:: ニューラル言語モデル) ができてしまいました

ニューラル言語モデルの人間離れした部分

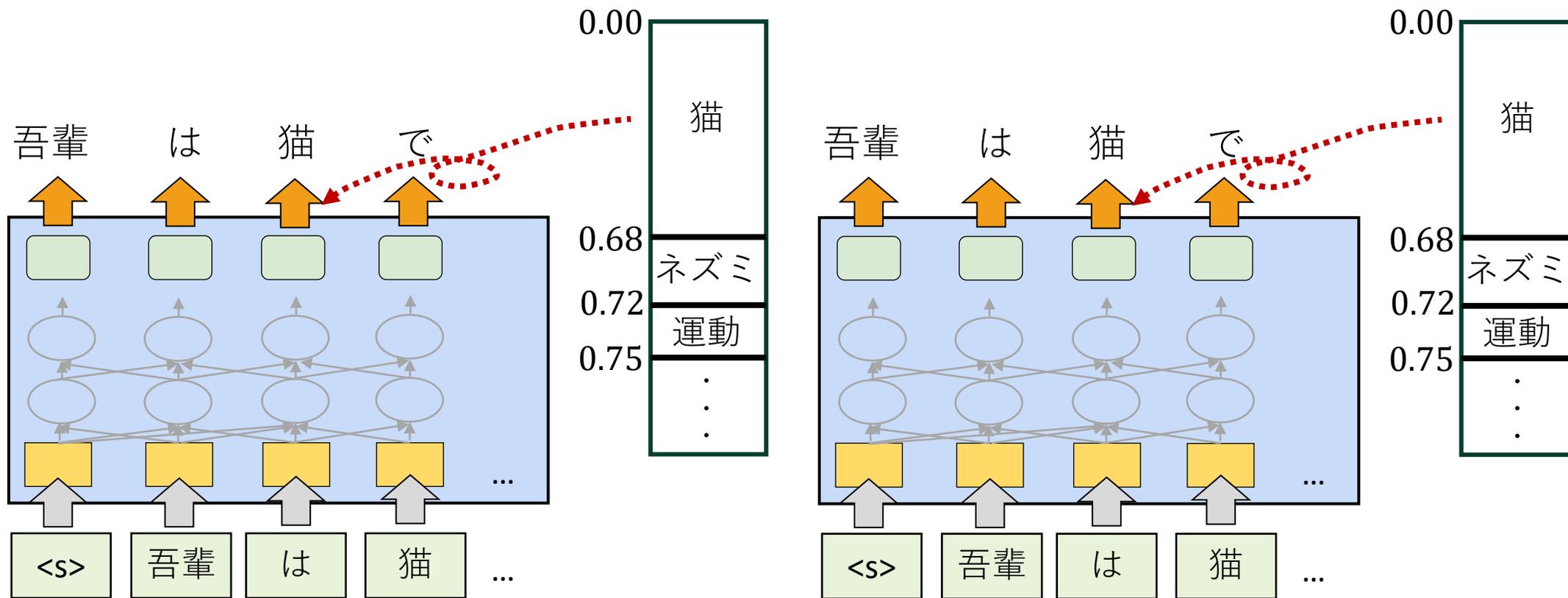


ニューラル言語モデルの人間離れした部分



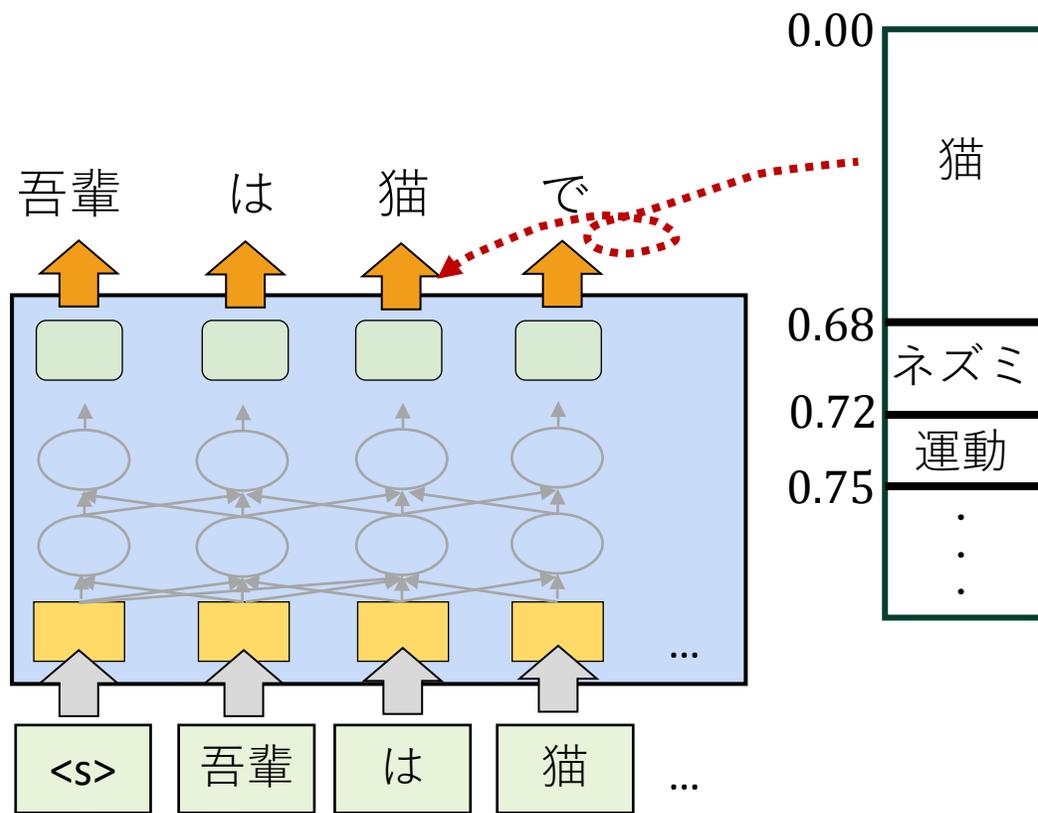
- 完全に同一のコピーが作れます

ニューラル言語モデルの人間離れした部分



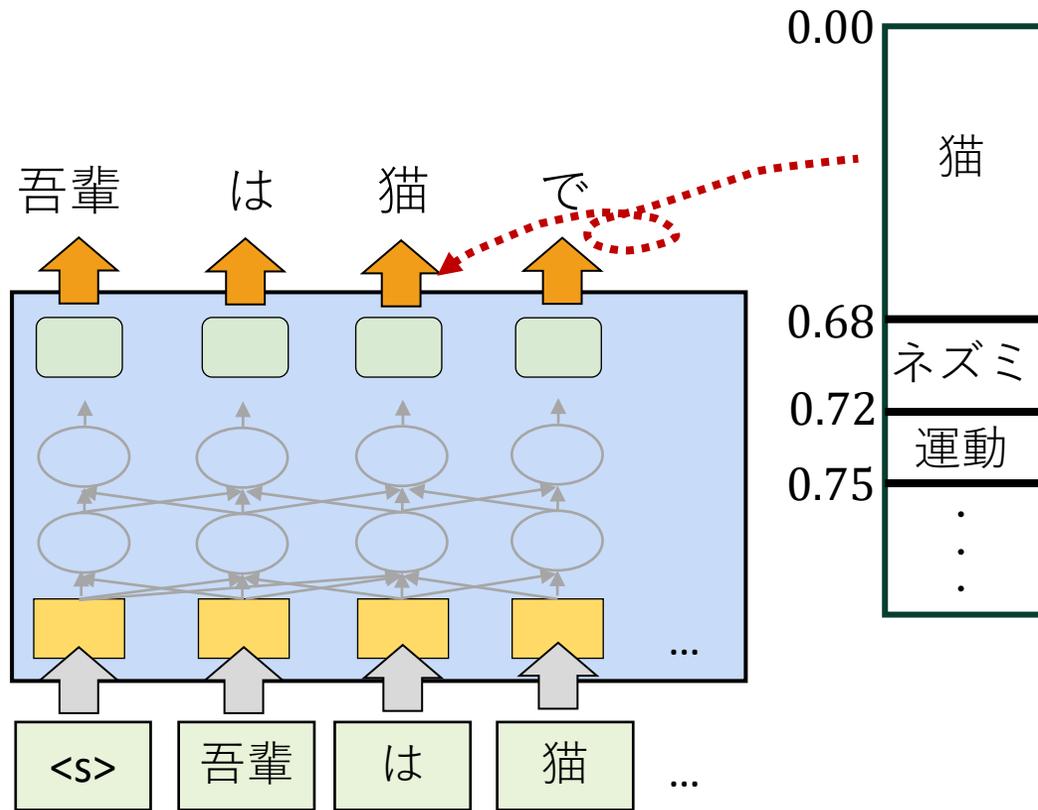
- 完全に同一のコピーが作れます
- 言語感覚は人によって異なるし、同一人物でも時間とともに変化します

ニューラル言語モデルの人間離れした部分



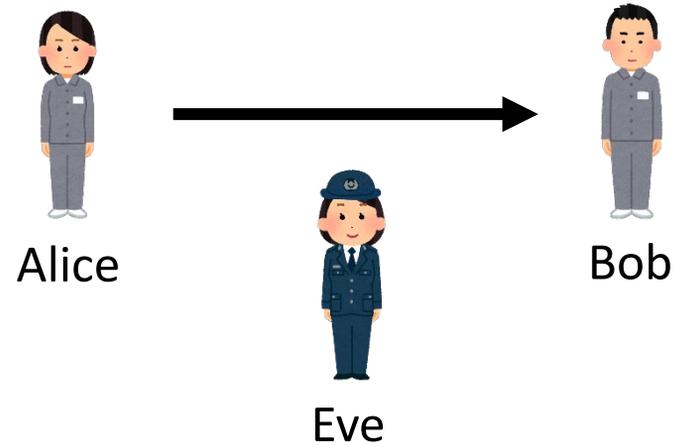
- 確率の値が外部から観測できます

ニューラル言語モデルの人間離れした部分



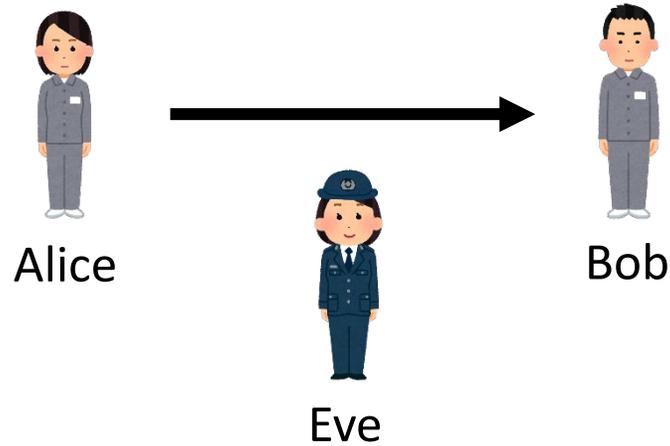
- 確率の値が外部から観測できます
- これら性質を応用して言語ステガノグラフィを実現します

ステガノグラフィ: 囚人の問題



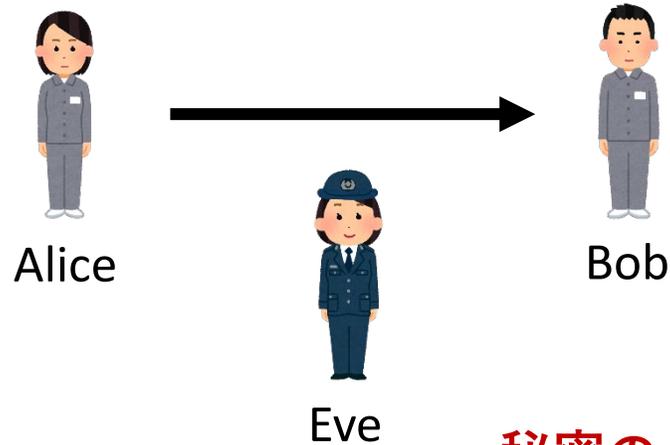
- 囚人**Alice**は囚人**Bob**に脱獄の計画を伝えたいが、看守**Eve**に常時監視されています
- **Eve**が不審に思わないように、何気ないテキストの中に秘密のメッセージを隠すことにします
- **Alice**と**Bob**は逮捕前にメッセージの隠し方を共有しておくとしています

ステガノグラフィ: 囚人の問題



- 例えば「今日は晴れているね」は「今夜脱獄」、「明日は曇りかな」は「翌朝脱獄」を表すと取り決めておきます

ステガノグラフィ: 囚人の問題

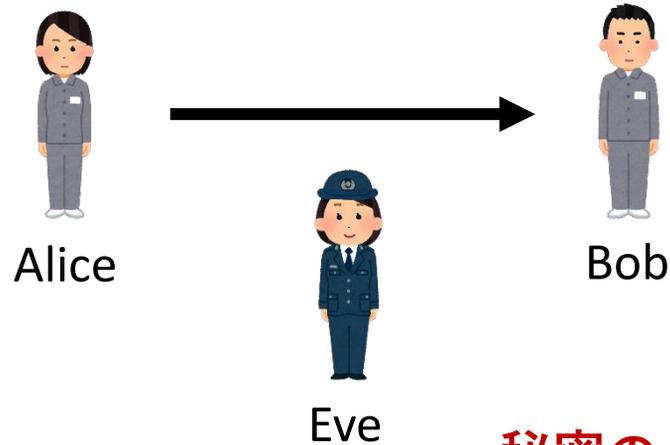


カバーテキスト

秘密のメッセージ

- 例えば「今日は晴れているね」は「今夜脱獄」、「明日は曇りかな」は「翌朝脱獄」を表すと取り決めておきます

ステガノグラフィ: 囚人の問題

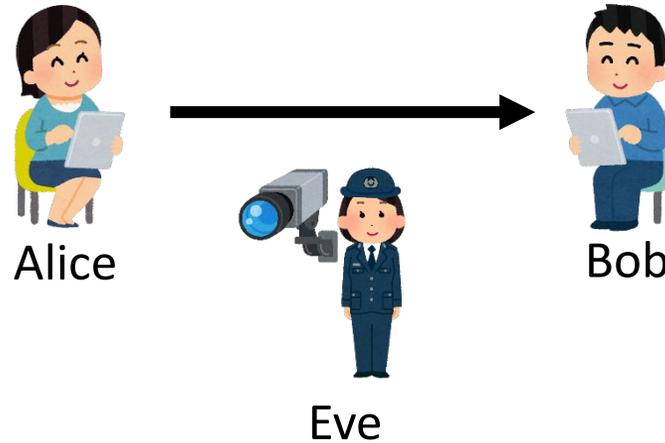


カバーテキスト

秘密のメッセージ

- 例えば「今日は晴れているね」は「今夜脱獄」、「明日は曇りかな」は「翌朝脱獄」を表すと取り決めておきます
- もっと柔軟にいろいろなメッセージを送れるようにするには？

ステガノグラフィ: コンピュータの利用



- Aliceはコンピュータを使って複雑なメッセージ埋め込み、Eveが監視する回線でBobに対して通信するとします
- デジタルデータは010011...のように0/1のビット列で表せるので、秘密のメッセージはビット列とします
- ビット列を埋め込んだ自然なカバーテキストを作るにはどうしたらいいでしょうか？

暗号とステガノグラフィの違い

- 暗号はEveが解読できないようにメッセージを変換する方法です
- 不審なメッセージが送られていることはEveにもわかります
- ステガノグラフィは自然なカバーデータ (この場合はテキスト) を送ることで、秘密のメッセージの存在自体を隠します

テキストをカバーデータに使うので 言語ステガノグラフィといます

- 一般には画像をカバーデータに使うことが多いです



何の変哲もない画像?

テキストをカバーデータに使うので 言語ステガノグラフィといます

- 一般には画像をカバーデータに使うことが多いです



何の変哲もない画像?

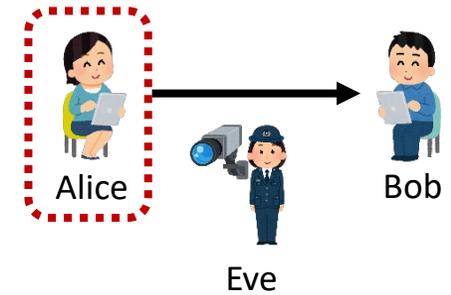


左の画像の色空間の低位ビットには別の画像が埋め込まれています!

言語のカバーデータとしての特性 (画像と比較した)

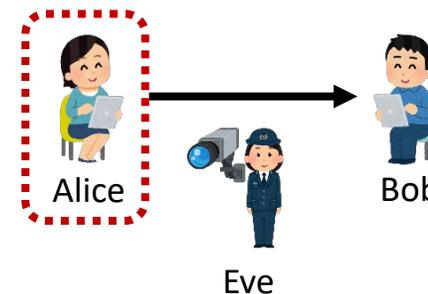
- 利点
 - 画像よりもテキストの方が気軽に作れるので、紛れ込ませやすい
- 欠点
 - 画像ほど冗長性がないので、少量のメッセージしか埋め込めない
 - 自然なテキストをコンピュータに作らせるのはこれまで難しかった

確率的言語モデルの利用: Aliceによるメッセージ埋め込み



秘密のメッセージ: 100001 ...
カバーテキスト: :

確率的言語モデルの利用: Aliceによるメッセージ埋め込み



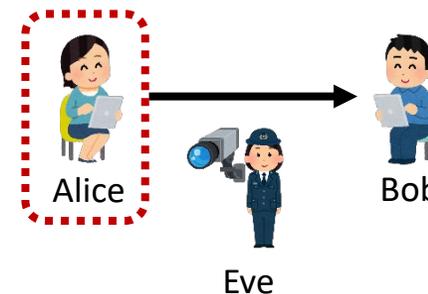
- 言語モデルに次の単語候補の確率を出力させます

秘密のメッセージ: 100001 ...
カバーテキスト: :

$P(w_1)$

単語	確率
今日	0.21
これ	0.20
私	0.15
また	0.13

確率的言語モデルの利用: Aliceによるメッセージ埋め込み



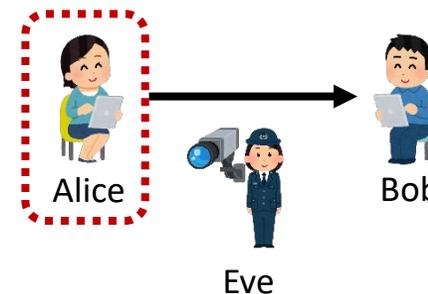
- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります

秘密のメッセージ: 100001 ...
カバーテキスト: :

$P(w_1)$

単語	確率	符号
今日	0.21	00
これ	0.20	01
私	0.15	10
また	0.13	11

確率的言語モデルの利用: Aliceによるメッセージ埋め込み



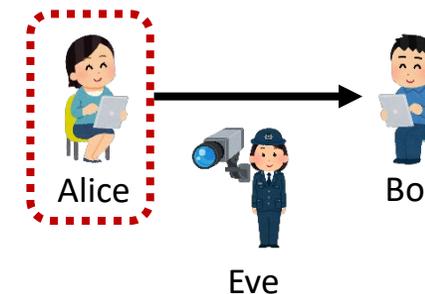
- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- 秘密のメッセージのビット列に対応する単語を選びます

秘密のメッセージ: **10**0001 ...
カバーテキスト: 私

$P(w_1)$

単語	確率	符号
今日	0.21	00
これ	0.20	01
私	0.15	10
また	0.13	11

確率的言語モデルの利用: Aliceによるメッセージ埋め込み

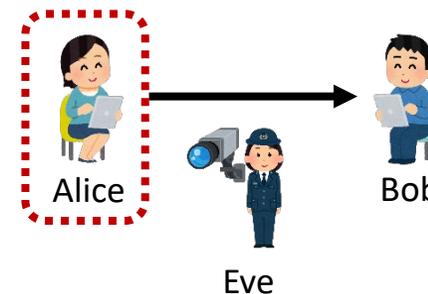


- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- 秘密のメッセージのビット列に対応する単語を選びます

秘密のメッセージ: 100001 ...
カバーテキスト: 私

$P(w_1)$			$P(w_2 w_1)$		
単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00
これ	0.20	01	たち	0.21	01
私	0.15	10	の	0.20	10
また	0.13	11	から	0.13	11

確率的言語モデルの利用: Aliceによるメッセージ埋め込み

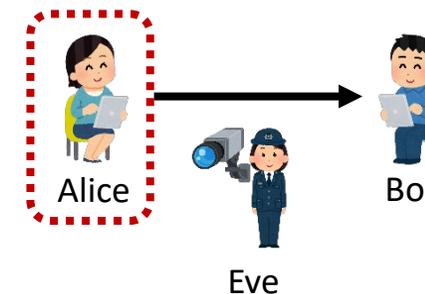


- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- 秘密のメッセージのビット列に対応する単語を選びます

秘密のメッセージ: 10**00**01 ...
カバーテキスト: 私 は

$P(w_1)$			$P(w_2 w_1)$		
単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00
これ	0.20	01	たち	0.21	01
私	0.15	10	の	0.20	10
また	0.13	11	から	0.13	11

確率的言語モデルの利用: Aliceによるメッセージ埋め込み

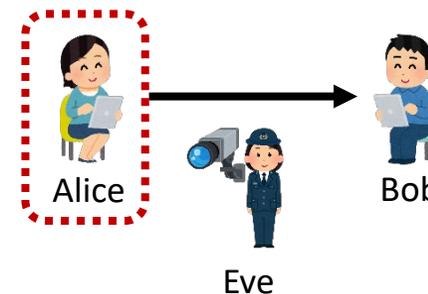


- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- 秘密のメッセージのビット列に対応する単語を選びます

秘密のメッセージ: 100001 ...
カバーテキスト: 私 は

$P(w_1)$			$P(w_2 w_1)$		
単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00
これ	0.20	01	たち	0.21	01
私	0.15	10	の	0.20	10
また	0.13	11	から	0.13	11

確率的言語モデルの利用: Aliceによるメッセージ埋め込み



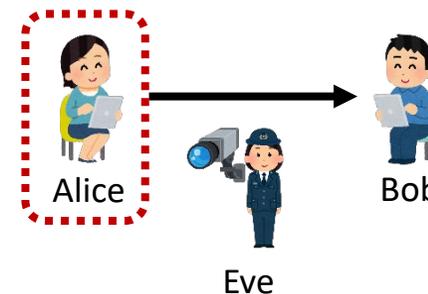
- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- 秘密のメッセージのビット列に対応する単語を選びます

秘密のメッセージ: 100001 ...
カバーテキスト: 私 は

$P(w_1)$			$P(w_2 w_1)$			$P(w_3 w_1, w_2)$		
単語	確率	符号	単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00	今日	0.35	00
これ	0.20	01	たち	0.21	01	また	0.25	01
私	0.15	10	の	0.20	10	もっと	0.12	10
また	0.13	11	から	0.13	11	つい	0.09	11

...

確率的言語モデルの利用: Aliceによるメッセージ埋め込み



- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- 秘密のメッセージのビット列に対応する単語を選びます

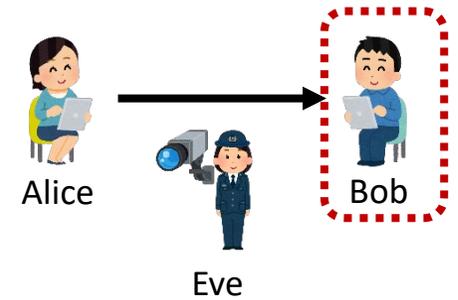
秘密のメッセージ: 1000**01**...

カバーテキスト: 私 は また ...

$P(w_1)$			$P(w_2 w_1)$			$P(w_3 w_1, w_2)$		
単語	確率	符号	単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00	今日	0.35	00
これ	0.20	01	たち	0.21	01	また	0.25	01
私	0.15	10	の	0.20	10	もっと	0.12	10
また	0.13	11	から	0.13	11	つい	0.09	11

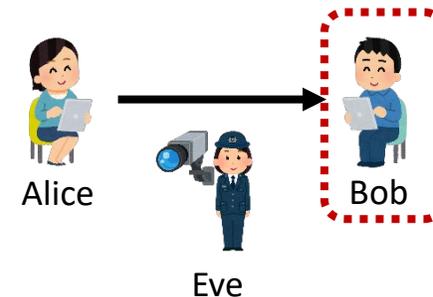
...

確率的言語モデルの利用: Bobによるメッセージ復元



カバーテキスト: 私 は また ...
秘密のメッセージ:

確率的言語モデルの利用: Bobによるメッセージ復元



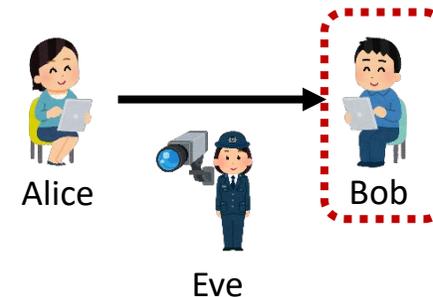
- 言語モデルに次の単語候補の確率を出力させます

カバーテキスト: 私 は また ...
秘密のメッセージ:

$P(w_1)$

単語	確率
今日	0.21
これ	0.20
私	0.15
また	0.13

確率的言語モデルの利用: Bobによるメッセージ復元



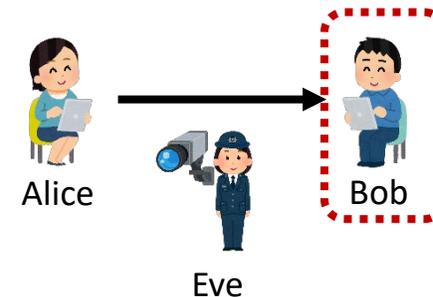
- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります

カバーテキスト: 私 は また ...
秘密のメッセージ:

$P(w_1)$

単語	確率	符号
今日	0.21	00
これ	0.20	01
私	0.15	10
また	0.13	11

確率的言語モデルの利用: Bobによるメッセージ復元



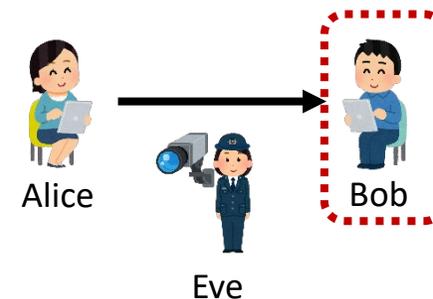
- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- カバーテキストの次の単語に対応するビット列を選びます

カバーテキスト: 私 は また ...
秘密のメッセージ: 10

$P(w_1)$

単語	確率	符号
今日	0.21	00
これ	0.20	01
私	0.15	10
また	0.13	11

確率的言語モデルの利用: Bobによるメッセージ復元

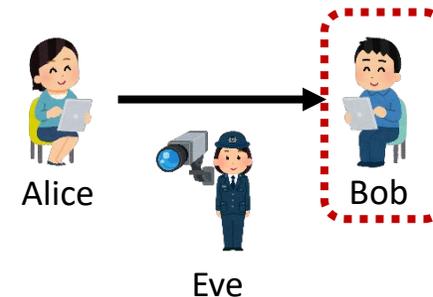


- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- カバーテキストの次の単語に対応するビット列を選びます

カバーテキスト: 私 は また ...
秘密のメッセージ: 10

$P(w_1)$			$P(w_2 w_1)$		
単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00
これ	0.20	01	たち	0.21	01
私	0.15	10	の	0.20	10
また	0.13	11	から	0.13	11

確率的言語モデルの利用: Bobによるメッセージ復元

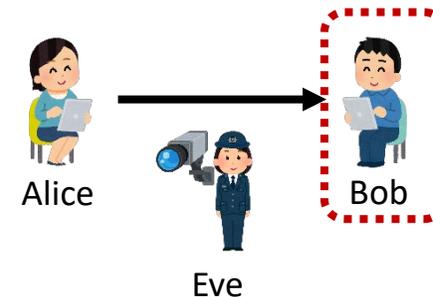


- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- カバーテキストの次の単語に対応するビット列を選びます

カバーテキスト: 私 **は** また ...
秘密のメッセージ: 1000

$P(w_1)$			$P(w_2 w_1)$		
単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00
これ	0.20	01	たち	0.21	01
私	0.15	10	の	0.20	10
また	0.13	11	から	0.13	11

確率的言語モデルの利用: Bobによるメッセージ復元



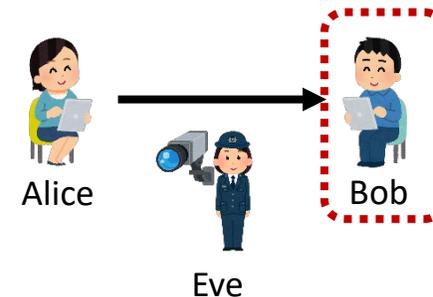
- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- カバーテキストの次の単語に対応するビット列を選びます

カバーテキスト: 私 は また ...
秘密のメッセージ: 1000

$P(w_1)$			$P(w_2 w_1)$			$P(w_3 w_1, w_2)$		
単語	確率	符号	単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00	今日	0.35	00
これ	0.20	01	たち	0.21	01	また	0.25	01
私	0.15	10	の	0.20	10	もっと	0.12	10
また	0.13	11	から	0.13	11	つい	0.09	11

...

確率的言語モデルの利用: Bobによるメッセージ復元



- 言語モデルに次の単語候補の確率を出力させます
- それらの単語にビット列を割り振ります
- カバーテキストの次の単語に対応するビット列を選びます

カバーテキスト: 私 は **また** ...
秘密のメッセージ: 100001 ...

$P(w_1)$			$P(w_2 w_1)$			$P(w_3 w_1, w_2)$		
単語	確率	符号	単語	確率	符号	単語	確率	符号
今日	0.21	00	は	0.25	00	今日	0.35	00
これ	0.20	01	たち	0.21	01	また	0.25	01
私	0.15	10	の	0.20	10	もっと	0.12	10
また	0.13	11	から	0.13	11	つい	0.09	11

...

デモ

デモ

The image shows a JupyterLab interface with two notebooks open side-by-side. The left notebook is titled 'encode.ipynb' and the right is 'decode.ipynb'. Both notebooks contain Python code for loading a model and performing operations on a prompt. The 'encode.ipynb' notebook has a cell with the following code:

```
[ ]: import himitsu

language = "ja"
device = "cpu"
min_prob = 0.01

model = himitsu.load_model(language, device)
tokenizer, byte_level_vocab, special_tokens = himitsu.

encode = lambda prompt, secret: himitsu.encode(
    model=model,
    tokenizer=tokenizer,
    secret=secret,
    prompt=prompt,
    min_prob=min_prob,
    special_tokens=special_tokens,
    byte_level_vocab=byte_level_vocab,
)

[ ]: encode("今日の名古屋は", "")
```

The 'decode.ipynb' notebook has a cell with the following code:

```
[ ]: import himitsu

language = "ja"
device = "cpu"
min_prob = 0.01

model = himitsu.load_model(language, device)
tokenizer, byte_level_vocab, special_tokens = himitsu.

decode = lambda prompt, cover_text: himitsu.decode(
    model=model,
    tokenizer=tokenizer,
    cover_text=cover_text,
    prompt=prompt,
    min_prob=min_prob,
    special_tokens=special_tokens,
    byte_level_vocab=byte_level_vocab,
)

[ ]: decode("今日の名古屋は", "")
```

The browser address bar shows 'localhost:10880/lab/tree/encode.ipynb'. The status bar at the bottom indicates 'Simple' mode, 'Python 3 (ipykernel) | Idle', and 'Mode: Edit Ln 5, Col 16 encode.ipynb 1'.

発展編1

```
tokenizer, byte_level_vocab, special_tokens = himitsu.

encode = lambda prompt, secret: himitsu.encode(
    model=model,
    tokenizer=tokenizer,
    secret=secret,
    prompt=prompt,
    min_prob=min_prob,
    special_tokens=special_tokens,
    byte_level_vocab=byte_level_vocab,
)

[2]: の名古屋は", "010000110101011011110001011010111110001"

[2]: '雨がっていますが、昨日までお天気の良い日に外仕事しましたが。'
```

```
decode = lambda prompt, cover_text: himitsu.decode(
    model=model,
    tokenizer=tokenizer,
    cover_text=cover_text,
    prompt=prompt,
    min_prob=min_prob,
    special_tokens=special_tokens,
    byte_level_vocab=byte_level_vocab,
)

[2]: decode("今日の名古屋は", "雨がっていますが、昨日までお天気

[2]: '0100001101010110111100010110101111100010'
```

- よく見ると末尾に余計な0がついています...
- 一般にメッセージ埋め込み終了時点とテキストの自然な終了位置は一致しません
- 解決策: メッセージの先頭部分にメッセージ長を載せることにすれば、`</s>`を出力するまで単語列を出力してもダミー部分が区別できます

発展編2: 秘密のメッセージとしてテキストを埋め込みたい

- 素朴な案: 文字コード表に従って各文字をビット列に変換
 - あ ⇔ 111000111000000110000010
 - ア ⇔ 111000111000001010100010

発展編2: 秘密のメッセージとしてテキストを埋め込みたい

- 素朴な案: 文字コード表に従って各文字をビット列に変換
 - あ ⇔ 111000111000000110000010
 - ア ⇔ 111000111000001010100010
- すごく効率が悪い...

発展編2: 秘密のメッセージとしてテキストを埋め込みたい

- 素朴な案: 文字コード表に従って各文字をビット列に変換
 - あ \Leftrightarrow 111000111000000110000010
 - ア \Leftrightarrow 111000111000001010100010
- すごく効率が悪い...
- 効率の良い変換表を作るには? 大学で情報理論を学びましょう

発展編3: 埋め込み効率を上げたい

$P(w_1)$			$P(w_2 w_1)$			$P(w_3 w_1, w_2)$			
単語	確率	符号	単語	確率	符号	単語	確率	符号	
今日	0.21	00	は	0.25	00	今日	0.35	00	
これ	0.20	01	たち	0.21	01	また	0.25	01	...
私	0.15	10	の	0.20	10	もっと	0.12	10	
また	0.13	11	から	0.13	11	つい	0.09	11	

- 1単語に2ビットより多く埋め込めない？

発展編3: 埋め込み効率を上げたい

$P(w_1)$			$P(w_2 w_1)$			$P(w_3 w_1, w_2)$			
単語	確率	符号	単語	確率	符号	単語	確率	符号	
今日	0.21	00	は	0.25	00	今日	0.35	00	...
これ	0.20	01	たち	0.21	01	また	0.25	01	
私	0.15	10	の	0.20	10	もっと	0.12	10	
また	0.13	11	から	0.13	11	つい	0.09	11	

- 1単語に2ビットより多く埋め込めない？
- これについても大学で情報理論を学びましょう

発展編4: 日本語は単語を区切らないけど...

- 単語が区切られたテキストは日本語としては不自然で、Eveが不審に思うかもしれない

発展編4: 日本語は単語を区切らないけど...

- 単語が区切られたテキストは日本語としては不自然で、Eveが不審に思うかもしれない
- もしAliceが単語区切りを消したテキストを送ったとすると、Bobは同じ単語区切りを常に再現できる？ ... No!

発展編4: 日本語は単語を区切らないけど...

- 単語が区切られたテキストは日本語としては不自然で、Eveが不審に思うかもしれない
- もしAliceが単語区切りを消したテキストを送ったとすると、Bobは同じ単語区切りを常に再現できる？ ... No!

東京都 ⇒ 東京 都
東 京都

ここにあった
画像は削除さ
れました

発展編4: 日本語は単語を区切らないけど...

- 単語が区切られたテキストは日本語としては不自然で、Eveが不審に思うかもしれない
- もしAliceが単語区切りを消したテキストを送ったとすると、Bobは同じ単語区切りを常に再現できる？ ... No!

東京都 ⇒ 東京 都
東 京都

ここにあった
画像は削除さ
れました

- 今回お見せしたデモはこの曖昧性問題に対処済みです

全体まとめ

- テキストを生成するには**1**単語ずつ出力すればよい
- 巨大なニューラル言語モデルは流暢にテキストを生成できます
- 言語モデルの人間離れした特性を応用すると、例えば秘密のメッセージを埋め込んだテキストが生成できます
- 言葉の研究にも意外と数学を使います

