
# COMP 3361 Natural Language Processing

## Lecture 7: Neural language models: Overview

Spring 2025

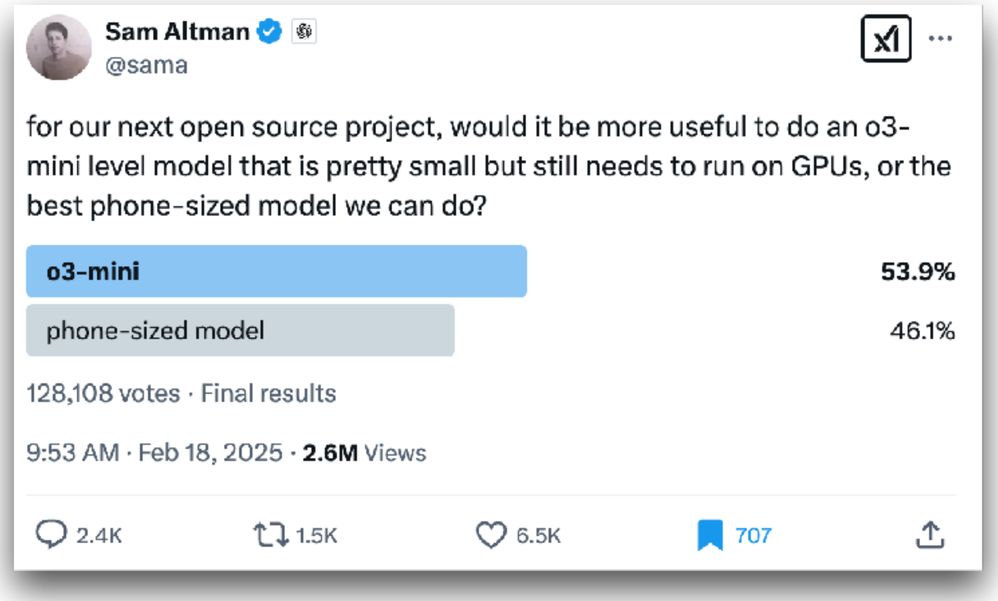
Many materials from CSE447@UW (Taylor Sorensen) and COS484@Princeton with special thanks!

# Announcements

- Assignment 1 is due on Mar 4!
  - Will provide a short coding tutorial next Friday
  - Book a TA slot via the link on the course page
  - Also you can always ask questions on Slack!
- We record a tutorial on PyTorch and Transformers as the make up lecture on Jan 28. Check it out on the course page.

# Latest AI news

- Grok 3 blog: Grok 3 Beta – The Age of Reasoning Agents
  - Try Grok 2 for free: https://x.com/i/grok
- OpenAI roadmap update for GPT-4.5 and GPT-5
- Qwen2.5-VL technical report is online



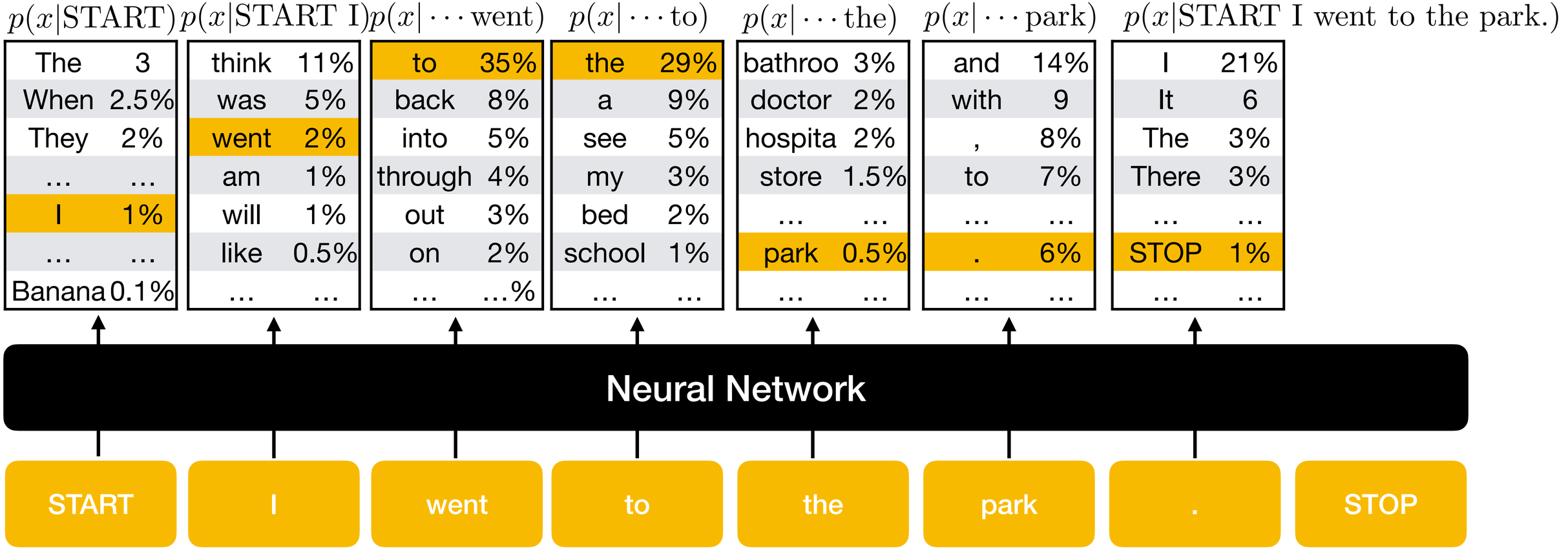
# Lecture plan

- Neural language models: overview
- Running examples of neural language models
- (Very quick) Deep learning review
- Language modeling with neural networks: RNNs

# Neural language models: overview

# Neural language models: inputs/outputs

- **Input:** sequences of words (or tokens)
- **Output:** probability distribution over the next word (token)



# Neural language models

*But neural networks take in real-valued vectors, not words…*

- Use one-hot or learned embeddings to map from words to vectors!
  - Learned embeddings become part of parameters $\theta$

*Neural networks output vectors, not probability distributions…*

- Apply the softmax to the outputs!
- What should the size of our output distribution be?
  - Same size as our vocabulary $|\mathcal{V}|$

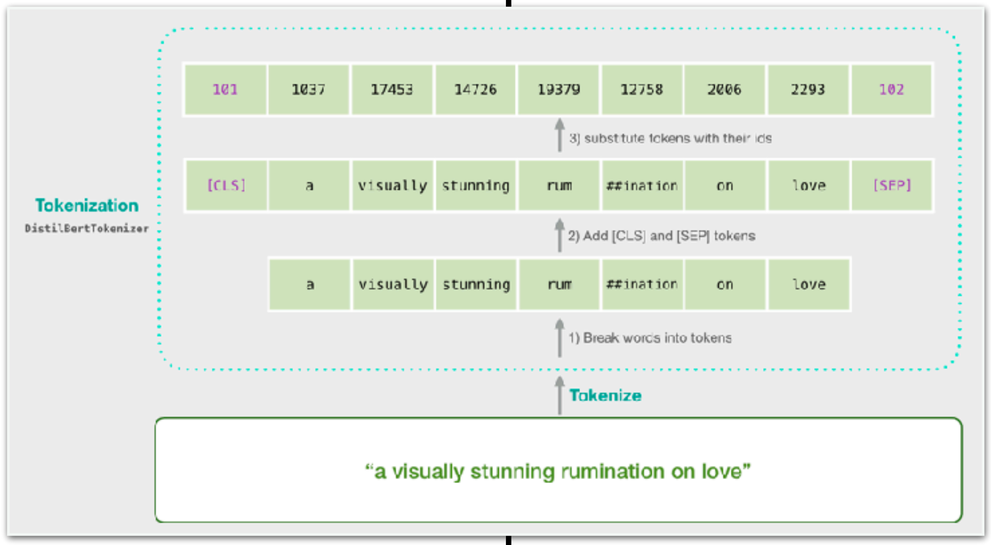
# Tokenization to input vectors

$p(x|\text{START})$ $p(x|\text{START I})$ $p(x|\cdots\text{went})$ $p(x|\cdots\text{to})$ $p(x|\cdots\text{the})$ $p(x|\cdots\text{park})$ $p(x|\text{START I went to the park.})$

**Neural Network**

Mapping each tokenized id into its corresponding embeddings

Tokenization:




START | I | went | to | the | park | . | STOP

# Example: BERT for sentiment classification

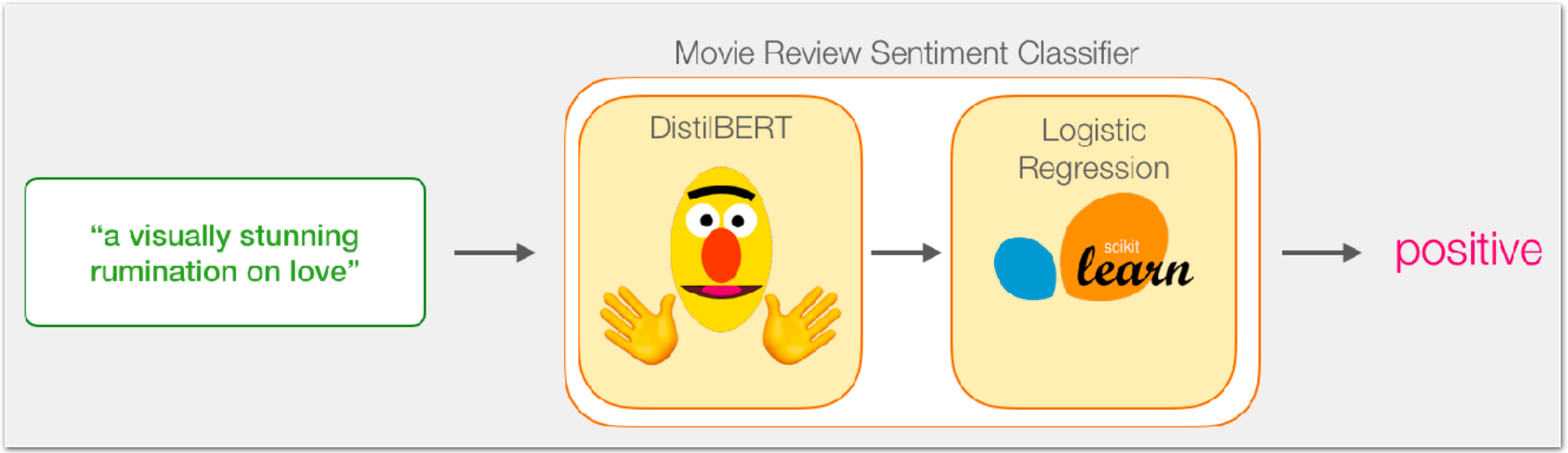
Task:


Data:

| sentence | label |
|---|---|
| a stirring , funny and finally transporting re imagining of beauty and the beast and 1930s horror films | 1 |
| apparently reassembled from the cutting room floor of any given daytime soap | 0 |
| they presume their audience won't sit still for a sociology lesson | 0 |
| this is a visually stunning rumination on love , memory , history and the war between art and commerce | 1 |
| jonathan parker 's bartleby should have been the be all end all of the modern office anomie films | 1 |

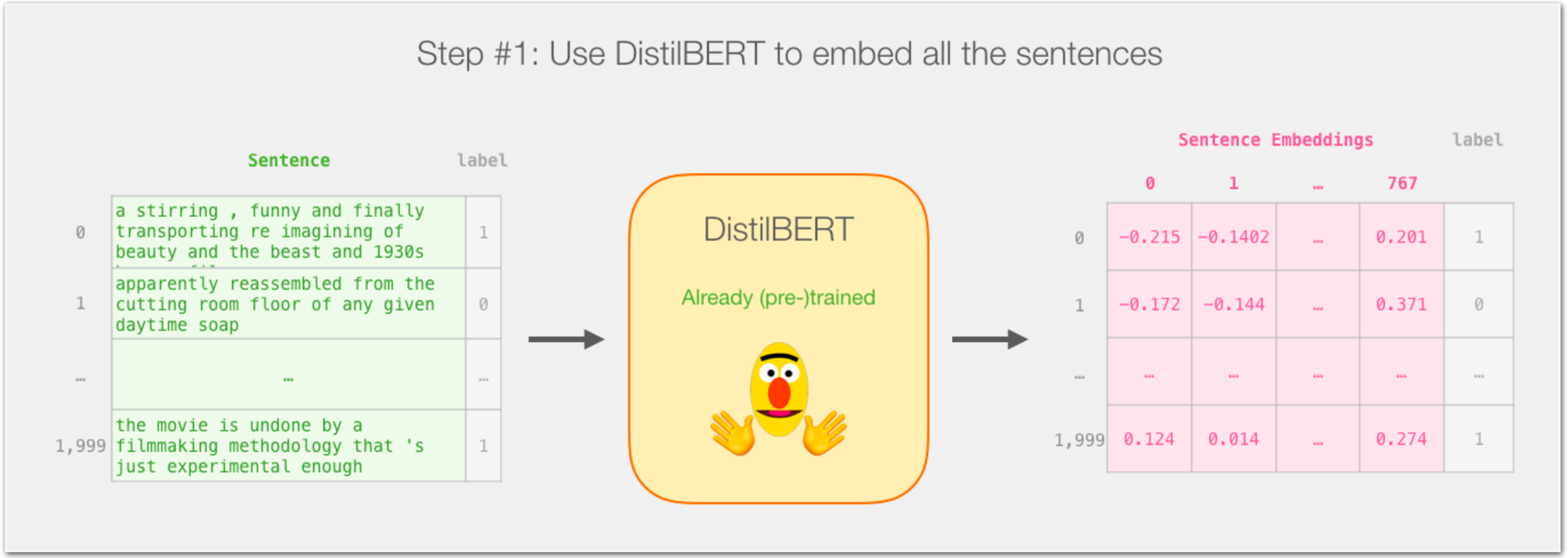
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

# Example: BERT for sentiment classification



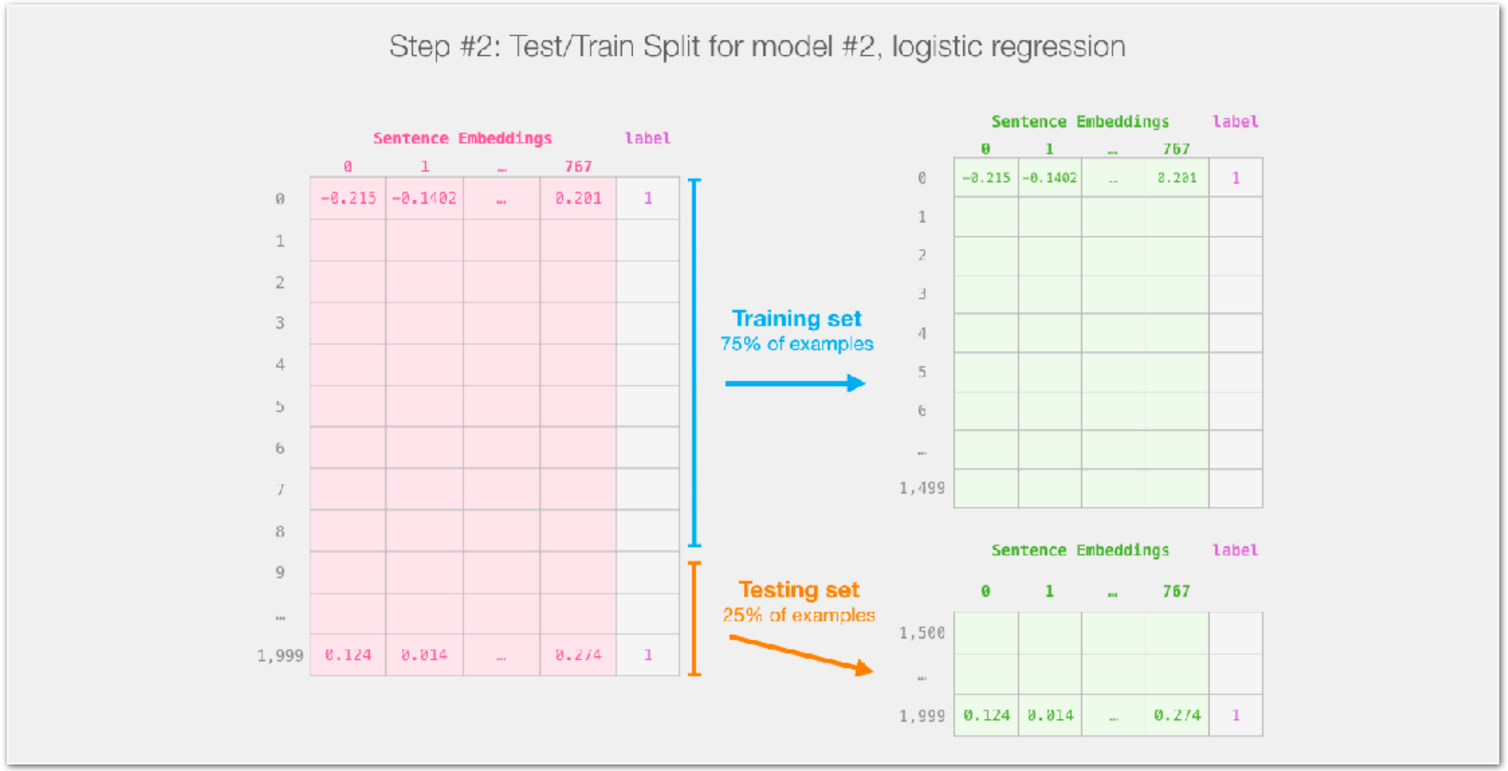

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

# BERT for sentiment classification: overview



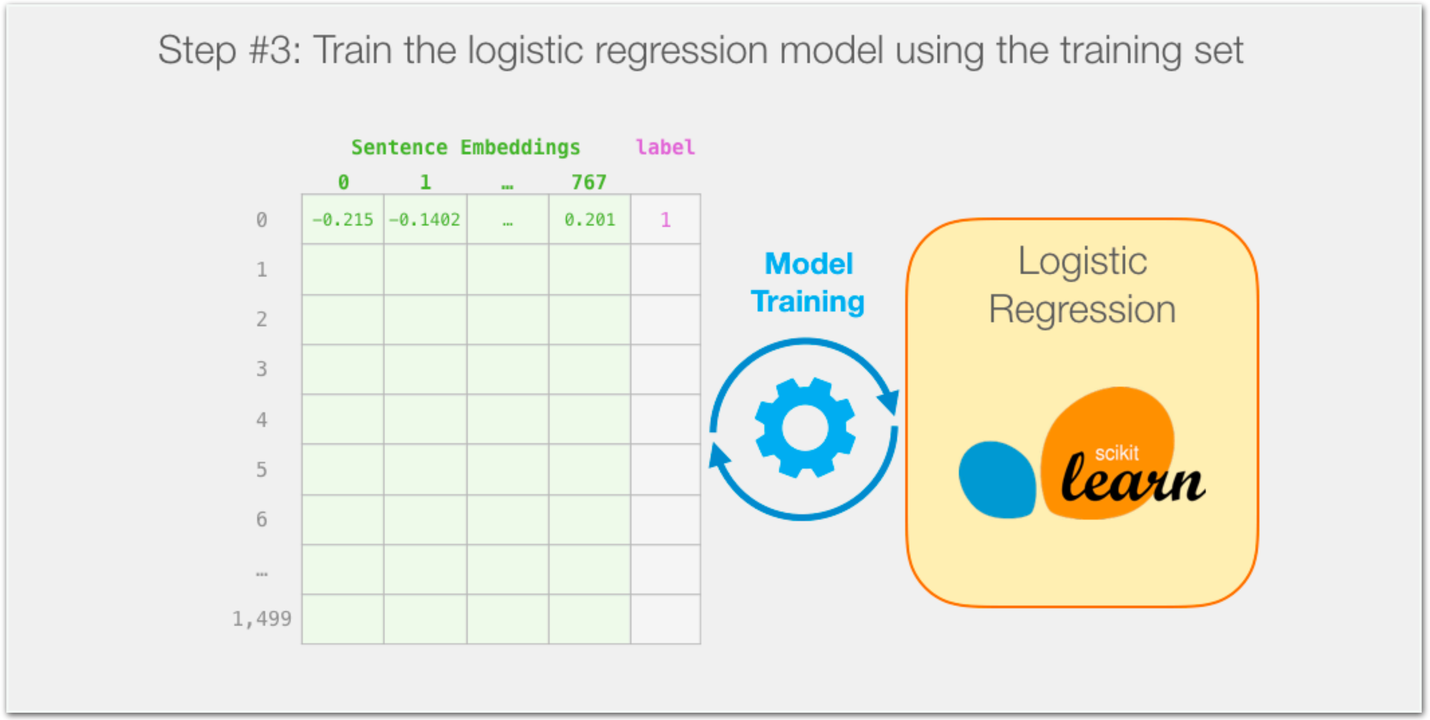

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

# BERT for sentiment classification: overview



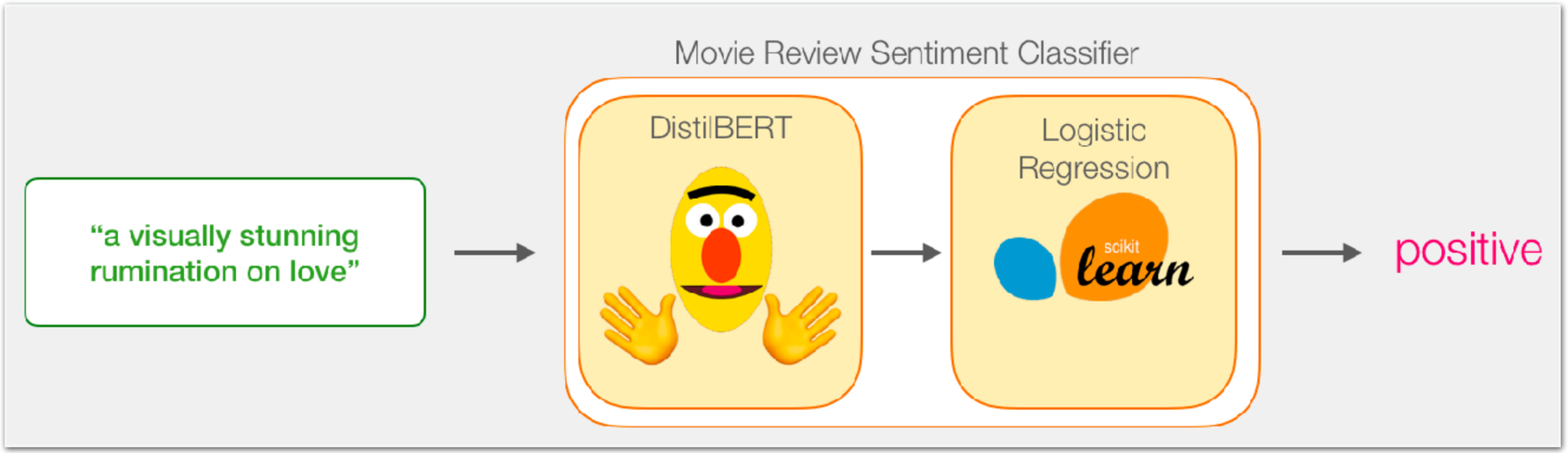

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

# BERT for sentiment classification: overview



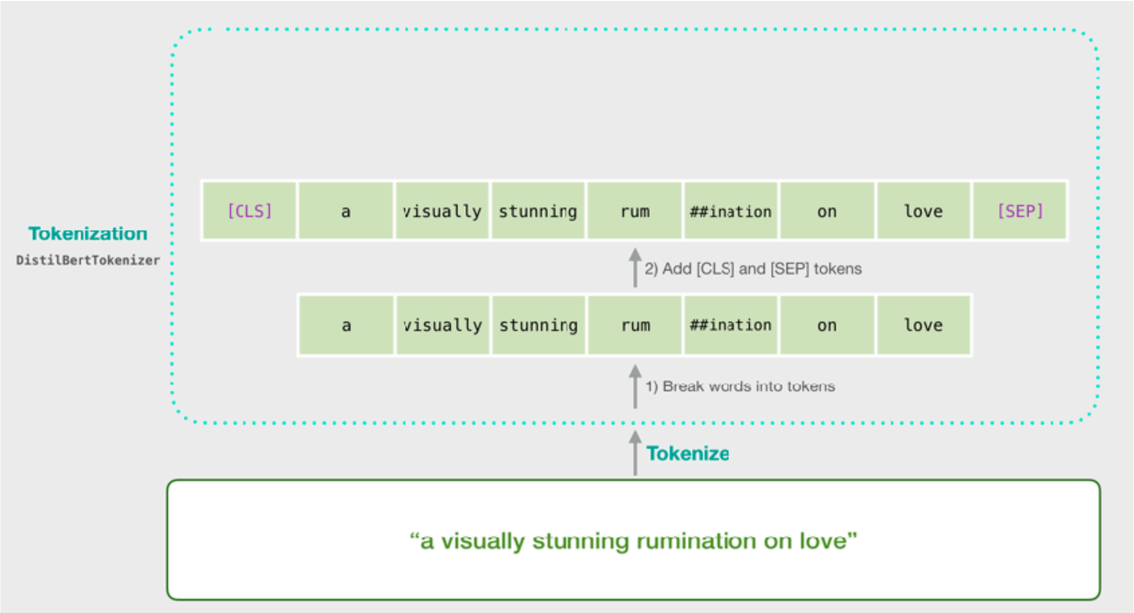

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

# BERT for sentiment classification: prediction



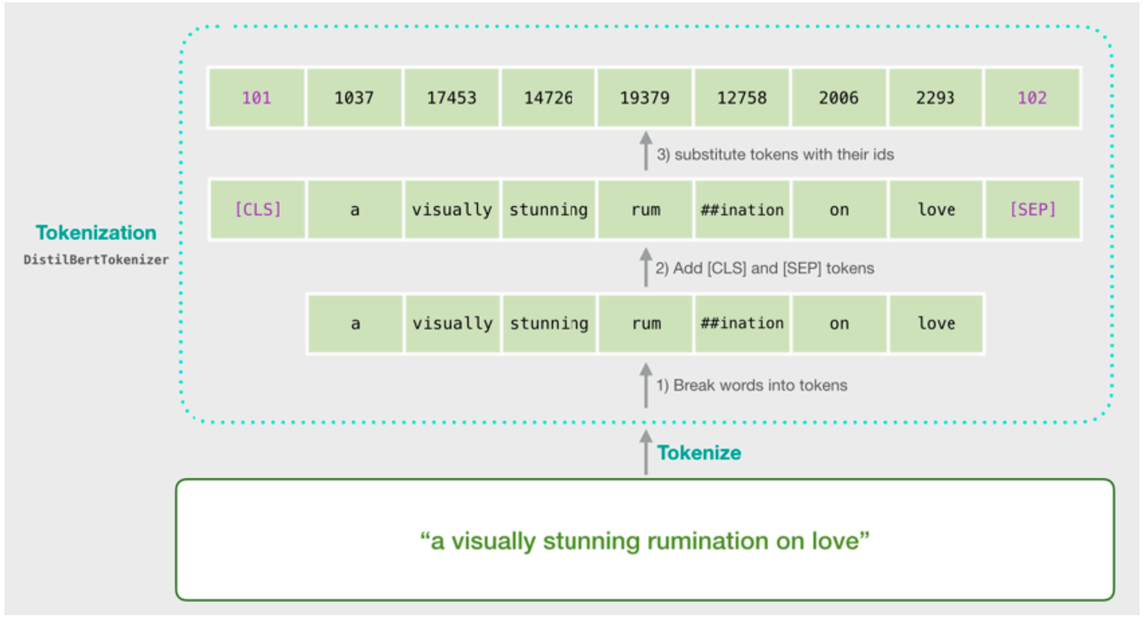

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

# Prediction step 1: tokenization




https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# Prediction step 1: tokenization




```
tokenized = df[0].apply((lambda x: tokenizer.encode(x, add_special_tokens=True)))
```

Raw Dataset

| 0 |
|---|
| a stirring , funny and finally transporting re... |
| apparently reassembled from the cutting room f... |
| they presume their audience wo n't sit still f... |
| this is a visually stunning rumination on love... |
| jonathan parker 's bartleby should have been t... |

Tokenize →

Sequences of Token IDs

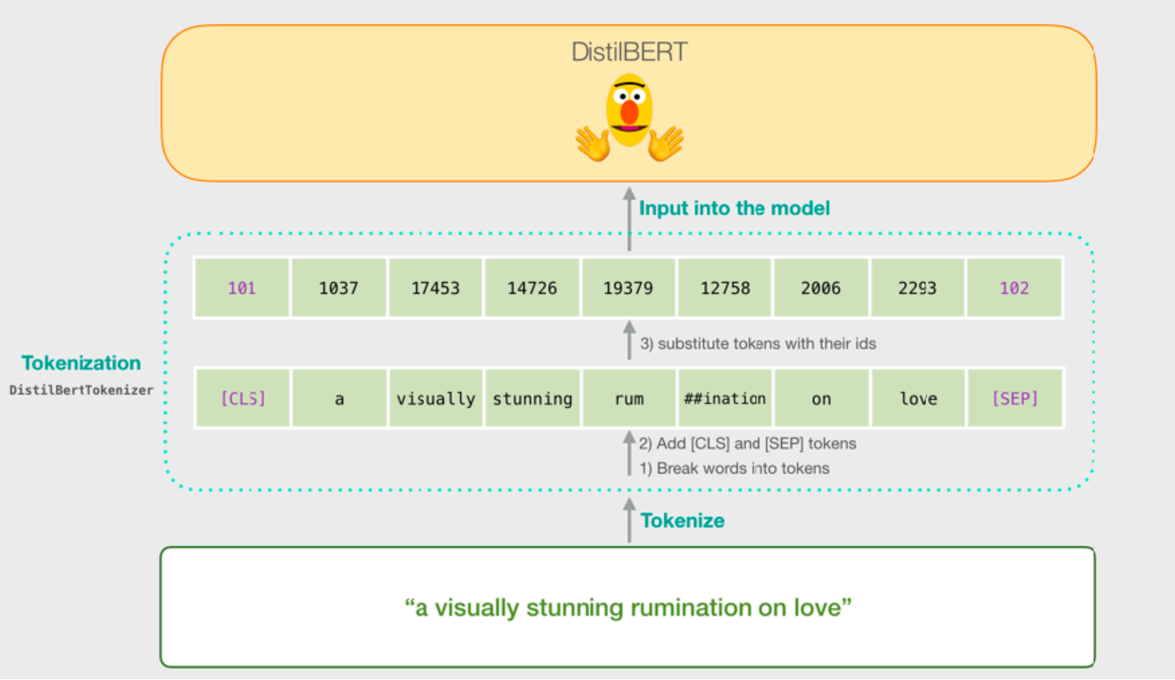
```
[101, 1037, 18385, 1010, 6057, 1998, 2633, 182...
[101, 4593, 2128, 27241, 23931, 2013, 1996, 62...
[101, 2027, 3653, 23545, 2037, 4378, 24185, 10...
[101, 2023, 2003, 1037, 17453, 14726, 19379, 1...
[101, 5655, 6262, 1005, 1055, 12075, 2571, 376...
```

BERT/DistilBERT Input Tensor

| | | Tokens in each sequence | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | … | 66 |
| Input sequences (reviews) | 0 | 101 | 1037 | … | 0 |
| | 1 | 101 | 2027 | … | 0 |
| | … | … | … | … | |
| | 1,999 | 101 | 1996 | … | 0 |

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# Prediction step 2: input into BERT




```
import numpy as np
import pandas as pd
import torch
import transformers as ppb # pytorch transformers
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
```

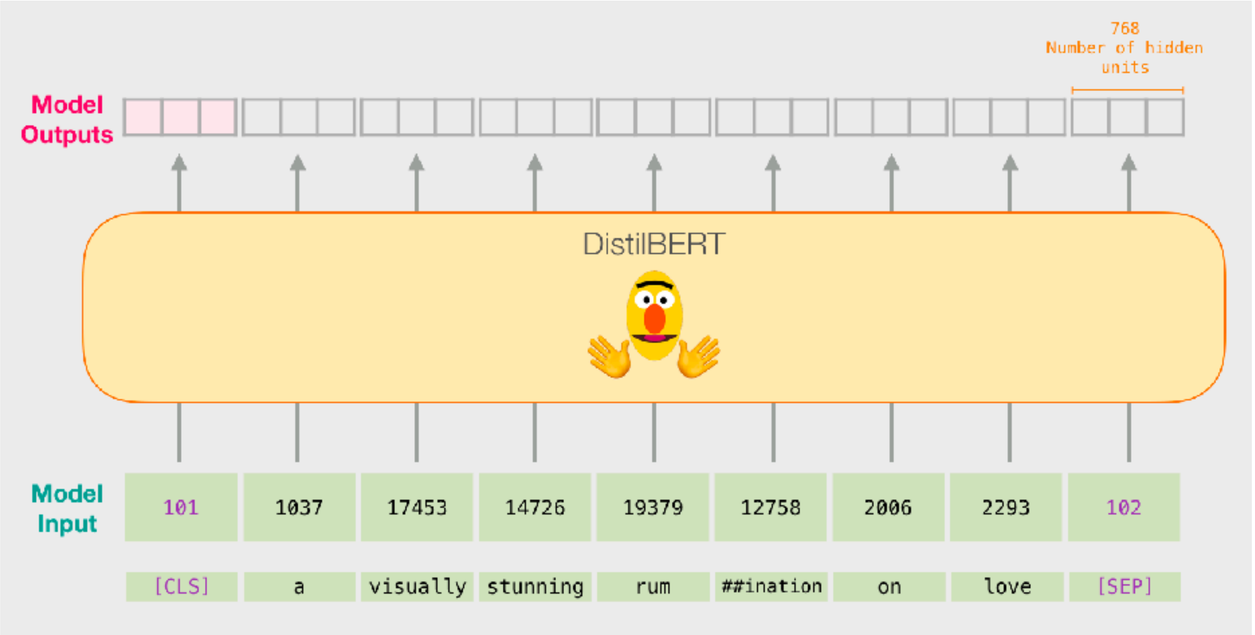
```
model_class, tokenizer_class, pretrained_weights = (ppb.DistilBertModel, ppb.DistilBertTokenizer, 'distilbert-base-uncased')

## Want BERT instead of distilBERT? Uncomment the following line:
#model_class, tokenizer_class, pretrained_weights = (ppb.BertModel, ppb.BertTokenizer, 'bert-base-uncased')

# Load pretrained model/tokenizer
tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
model = model_class.from_pretrained(pretrained_weights)
```

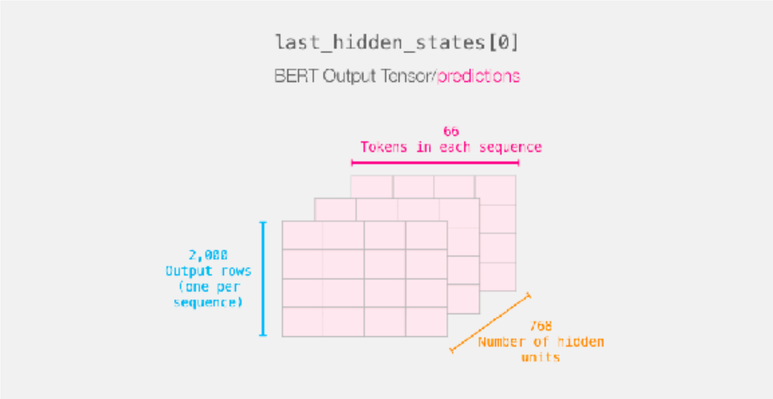
https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# Prediction step 3: run BERT to get outputs



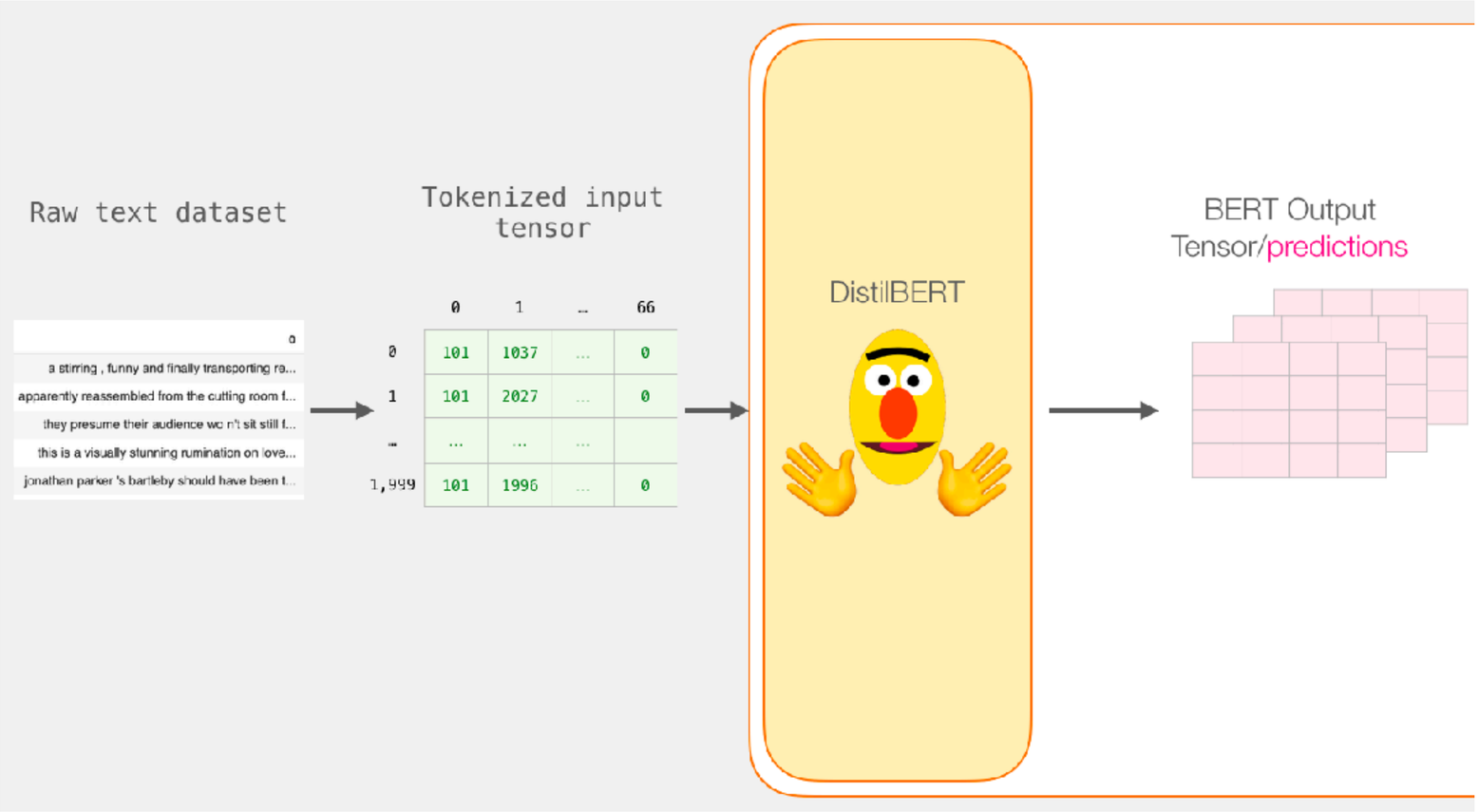

```
input_ids = torch.tensor(np.array(padded))

with torch.no_grad():
    last_hidden_states = model(input_ids)
```



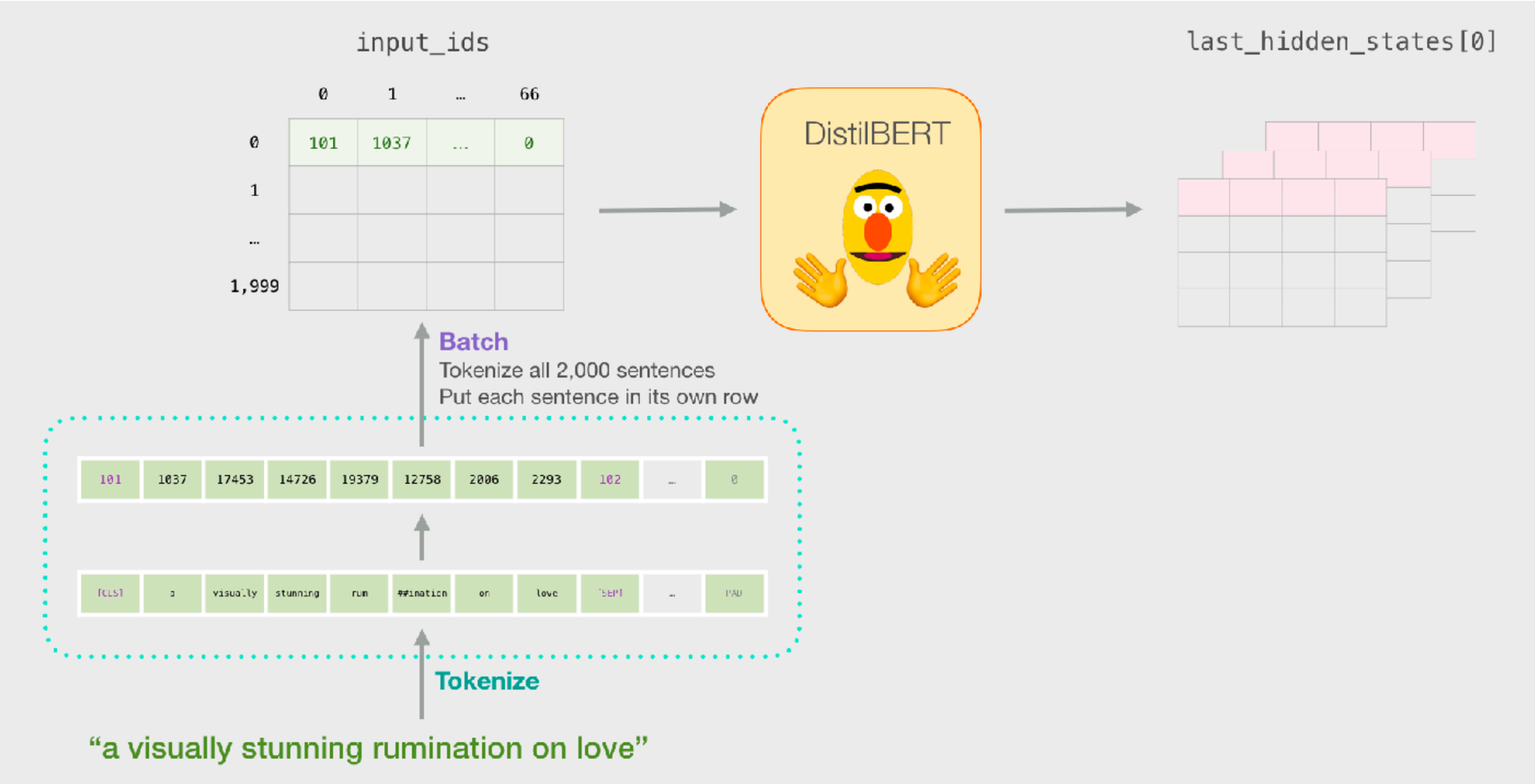

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# Example overview so far



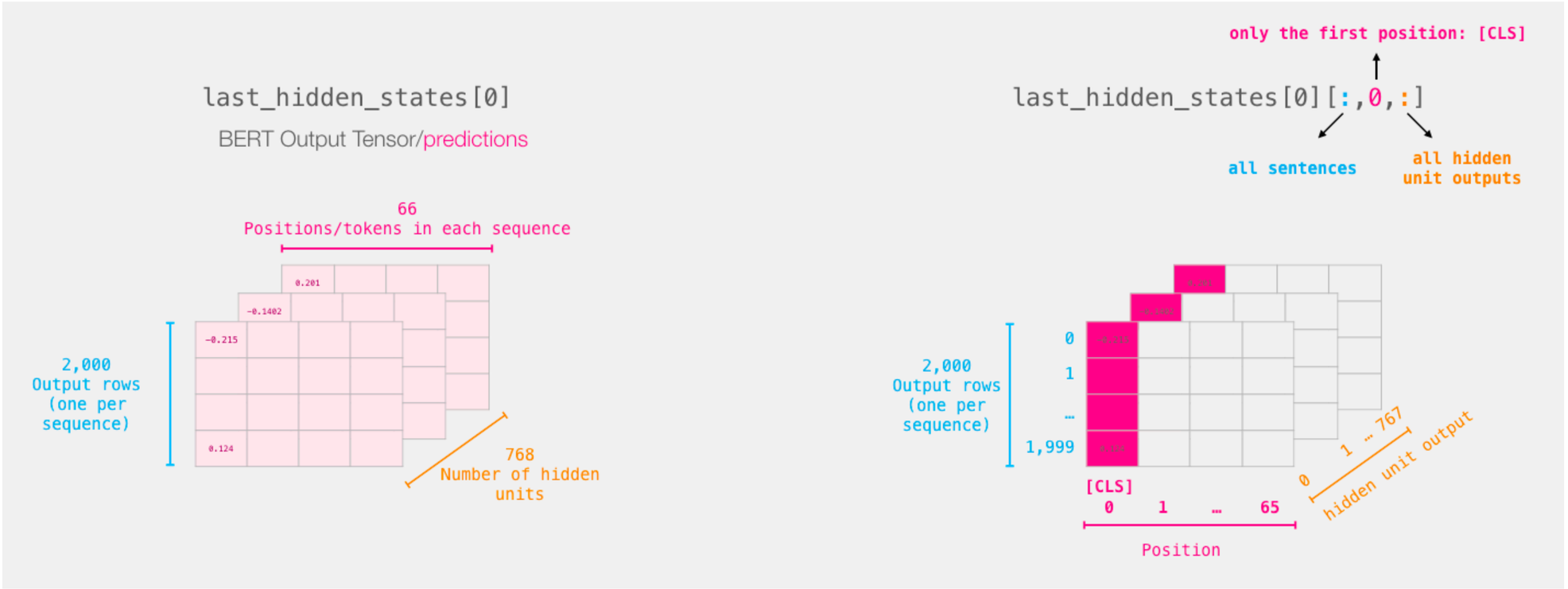

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# Recapping a sentence's journey




https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

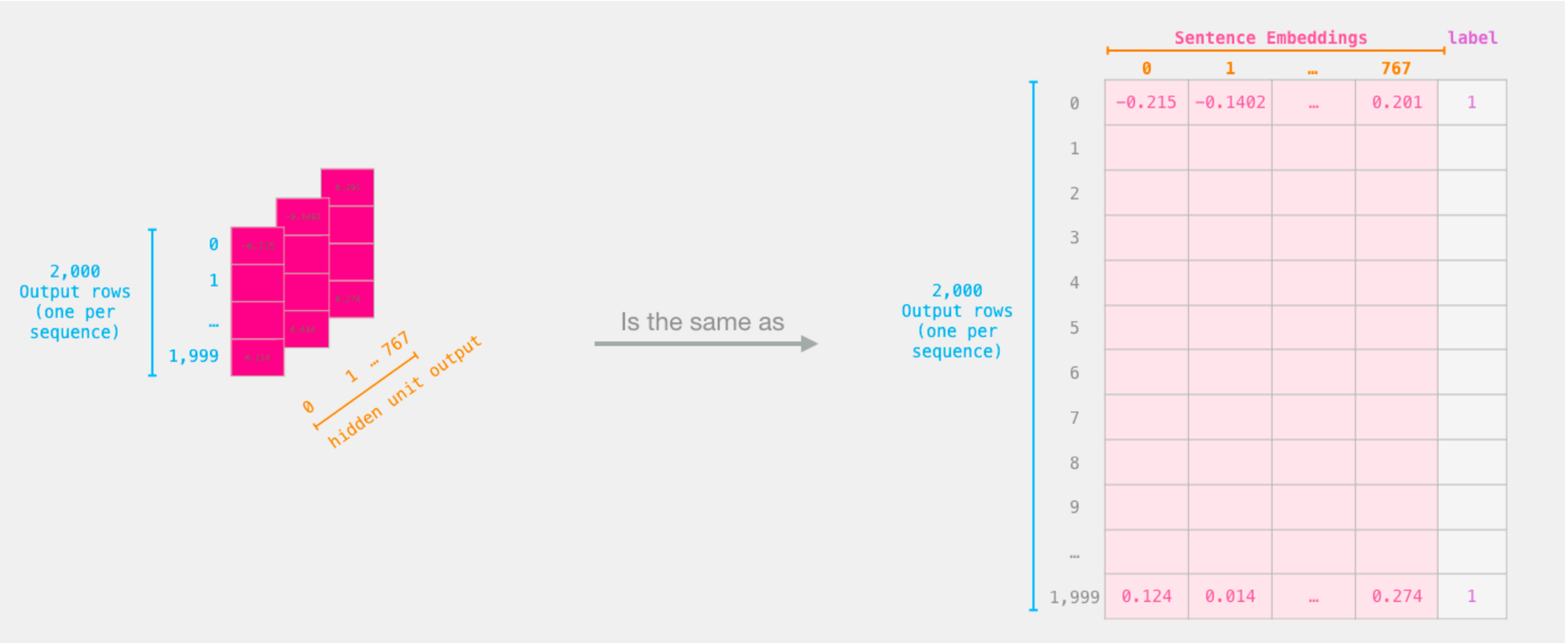
# Slicing the important part




```
# Slice the output for the first position for all the sequences, take all hidden unit outputs
features = last_hidden_states[0][:,0,:].numpy()
```

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

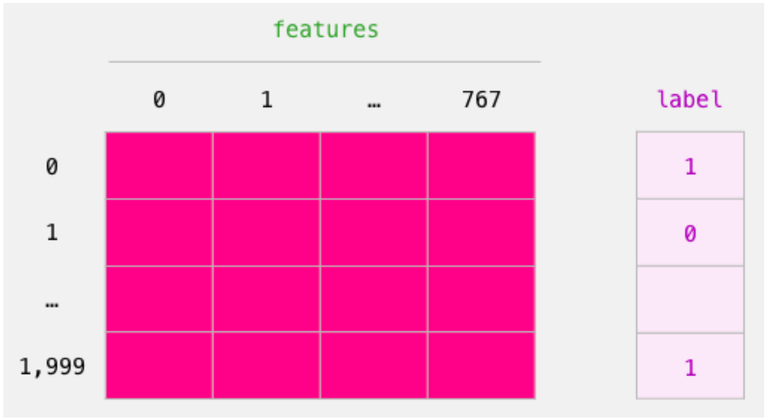
# Final BERT output features

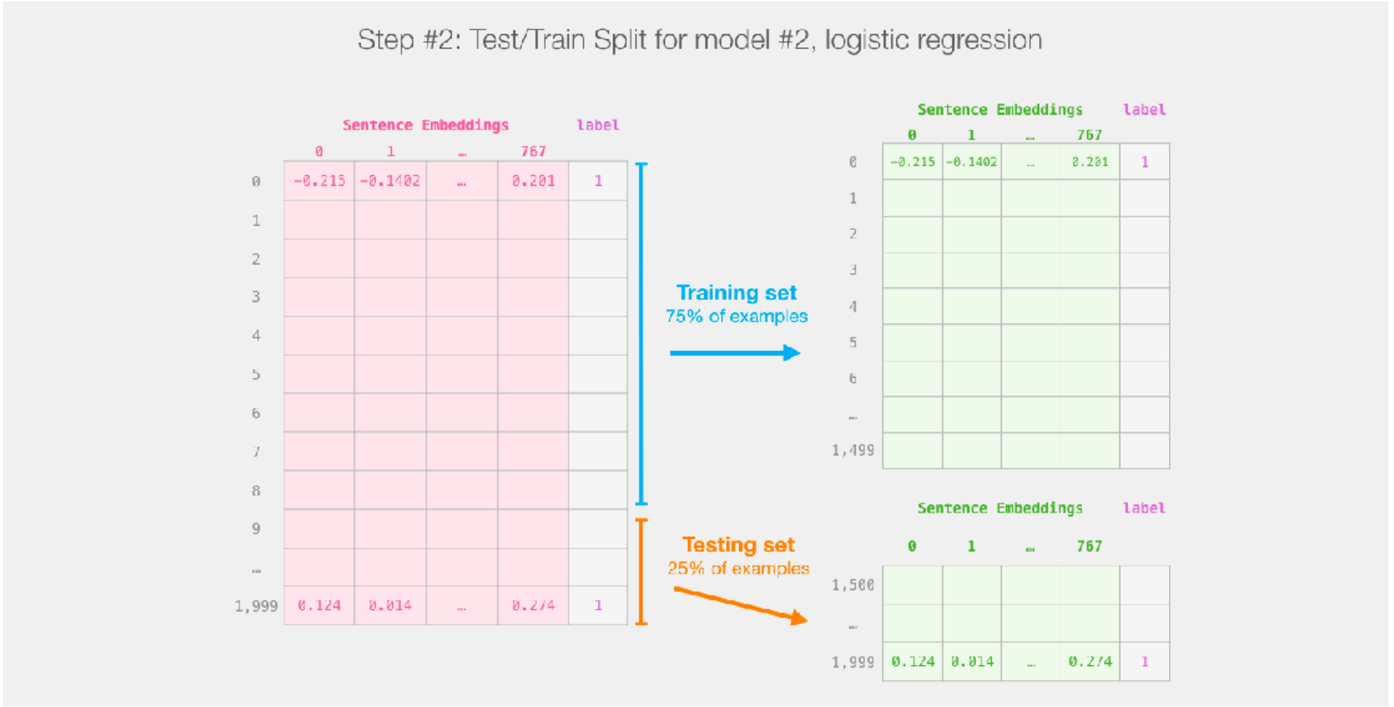



```
# Slice the output for the first position for all the sequences, take all hidden unit outputs
features = last_hidden_states[0][:,0,:].numpy()
```

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# Dataset for logistic regression

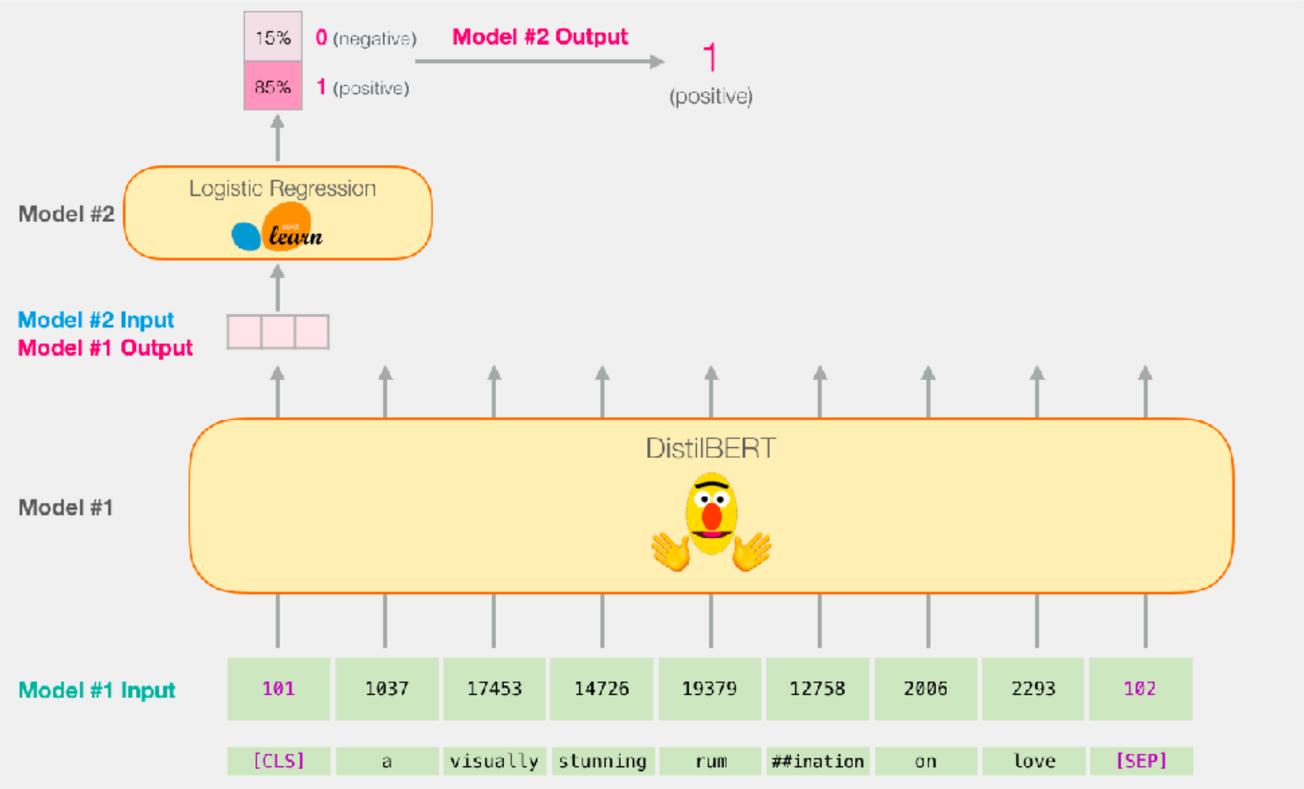



```
labels = df[1]
train_features, test_features, train_labels, test_labels = train_test_split(features, labels)
```

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# Prediction step 4: get final predictions




Train a logistic regression classifier

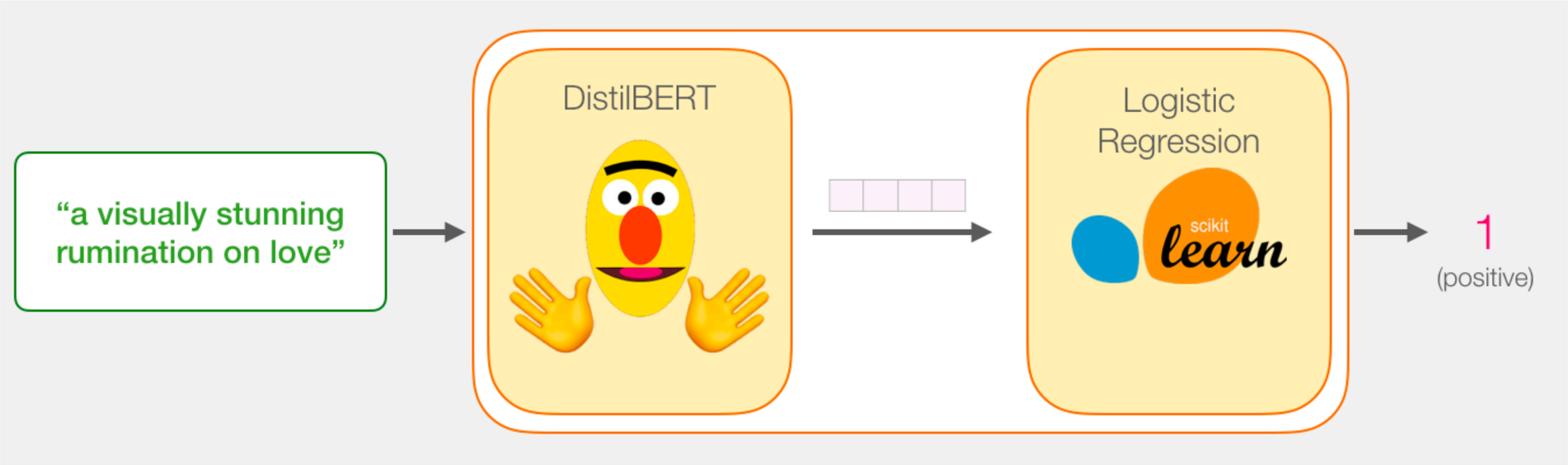
```
lr_clf = LogisticRegression()
lr_clf.fit(train_features, train_labels)
```

Run the trained logistic regression classifier

```
lr_clf.score(test_features, test_labels)
```

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# Example overview: BERT for sentiment classification




https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/, run the code on Colab!

# BERT is a encoder-only language model

**BERT (Bidirectional Encoder Representations from Transformers)**

1. **Encoder-Only Model**: BERT is designed as an encoder-only model. In the context of the Transformer architecture, an encoder processes the input data (like text) to create a representation of it. BERT's architecture is composed entirely of such encoders.
2. **Cannot Generate Words**: BERT is not designed to generate text in the same way models like GPT-3 do. Instead, its strength lies in understanding the context and meaning of words in a sentence. This is why it excels in tasks like sentence classification, entity recognition, and question-answering, where understanding context is crucial.
3. **Working Mechanism**: BERT analyzes and encodes the input text, taking into account both the left and right context of each word in the input. This bidirectional understanding is a key feature that differentiates BERT from earlier models that could only analyze text in a single direction.

Credits: ChatGPT

# How about BERT vs. GPT-3?

### BERT (Bidirectional Encoder Representations from Transformers)

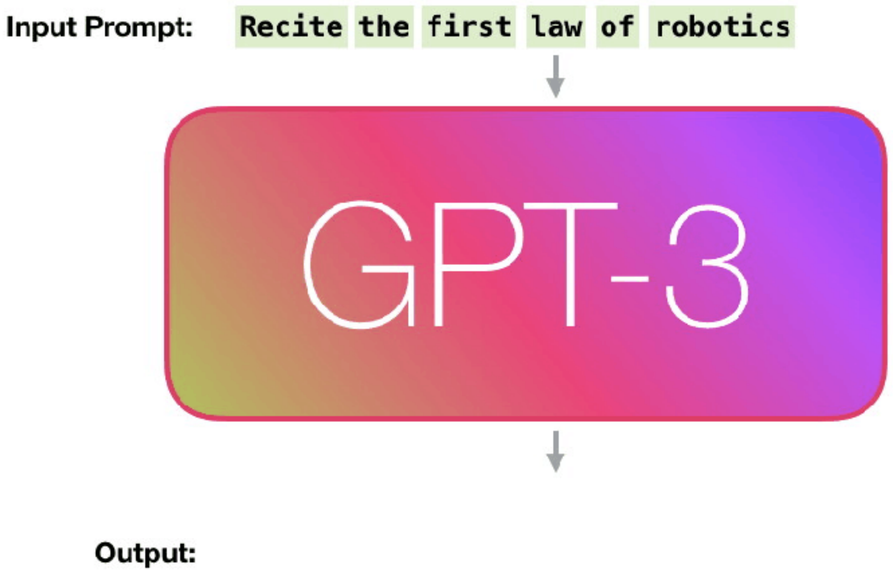
1. **Encoder-Only Model**: BERT is designed as an encoder-only model. In the context of the Transformer architecture, an encoder processes the input data (like text) to create a representation of it. BERT's architecture is composed entirely of such encoders.
2. **Cannot Generate Words**: BERT is not designed to generate text in the same way models like GPT-3 do. Instead, its strength lies in understanding the context and meaning of words in a sentence. This is why it excels in tasks like sentence classification, entity recognition, and question-answering, where understanding context is crucial.
3. **Working Mechanism**: BERT analyzes and encodes the input text, taking into account both the left and right context of each word in the input. This bidirectional understanding is a key feature that differentiates BERT from earlier models that could only analyze text in a single direction.

### GPT-3 (Generative Pretrained Transformer 3)

1. **Decoder-Only Model**: GPT-3, on the other hand, is a decoder-only model. In the Transformer architecture, a decoder is designed to generate output based on the input it receives. While BERT focuses on understanding and encoding input, GPT-3 focuses on generating output.
2. **Generates Tokens**: GPT-3 is capable of generating text, making it suitable for tasks like text completion, creative writing, and dialogue generation. It generates one word (or token) at a time and can continue generating text based on the context provided by the previous text.
3. **Working Mechanism**: GPT-3 uses a unidirectional approach, meaning it only considers the context to the left (previous tokens) when generating a new token. This design is conducive to generating coherent and contextually relevant continuations of the input text.

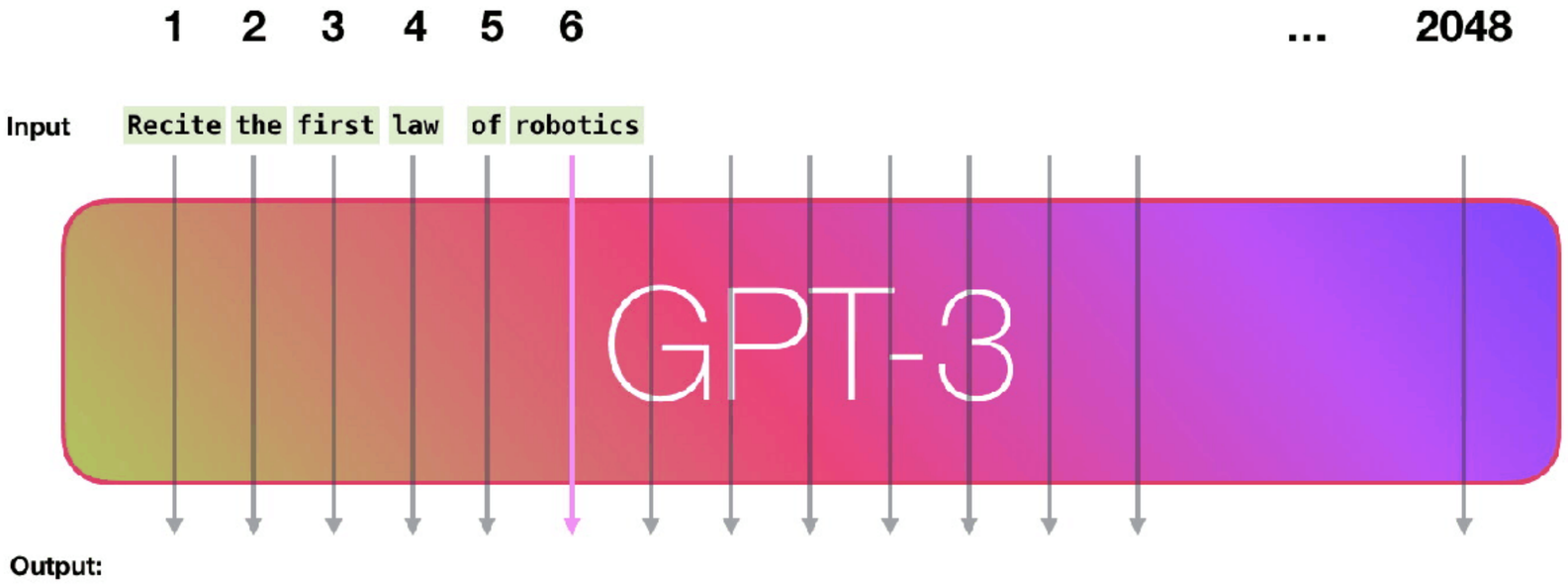
Credits: ChatGPT

# Usage example: GPT-3



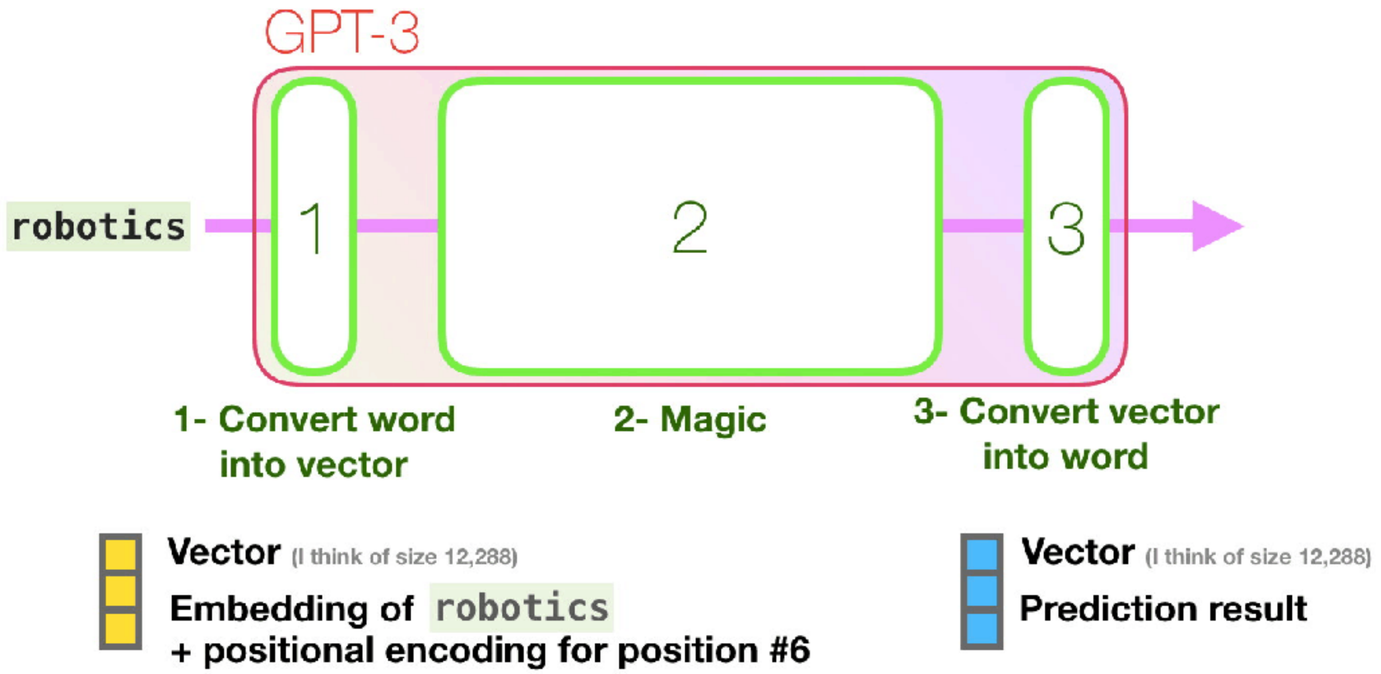

https://jalammar.github.io/how-gpt3-works-visualizations-animations/

# Usage example: GPT-3




https://jalammar.github.io/how-gpt3-works-visualizations-animations/

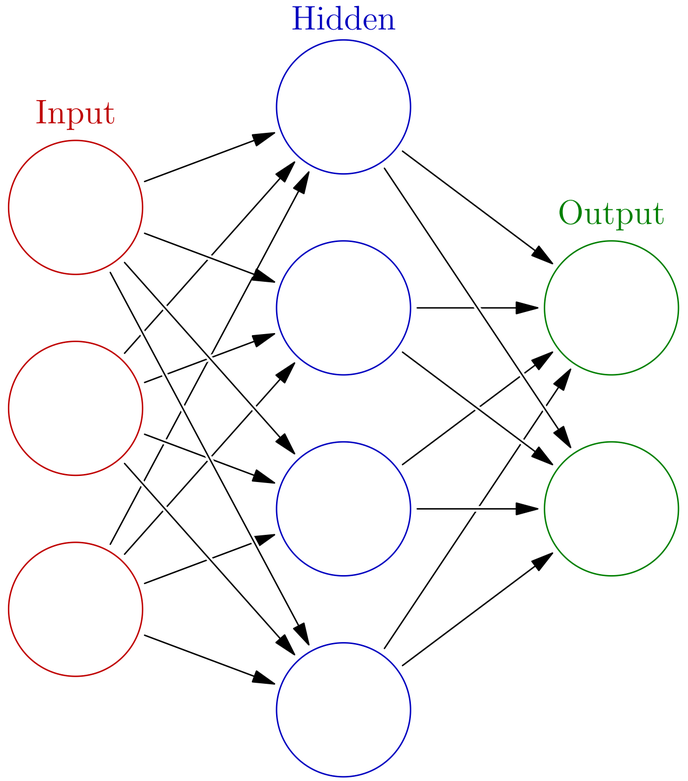
# Usage example: GPT-3




https://jalammar.github.io/how-gpt3-works-visualizations-animations/

# (Very quick) Deep learning review

# Neural networks




**Goal:** Approximate some function $f : \mathbb{R}^k \rightarrow \mathbb{R}^d$

**Essential elements:**

- Input: Vector $x \in \mathbb{R}^k$, Output: $y \in \mathbb{R}^d$
- Hidden representation layers $h_i \in \mathbb{R}^{d_i}$
- Non-linear, differentiable (almost everywhere) activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ (applied element-wise)
- Weights connecting layers: $W \in \mathbb{R}^{d_{i+1} \times d_i}$ and bias term $b \in \mathbb{R}^{d_{i+1}}$
  - Set of all parameters is often referred to as $\theta$

$$\hat{y} = W_2\sigma(W_1 x + b_1) + b_2$$

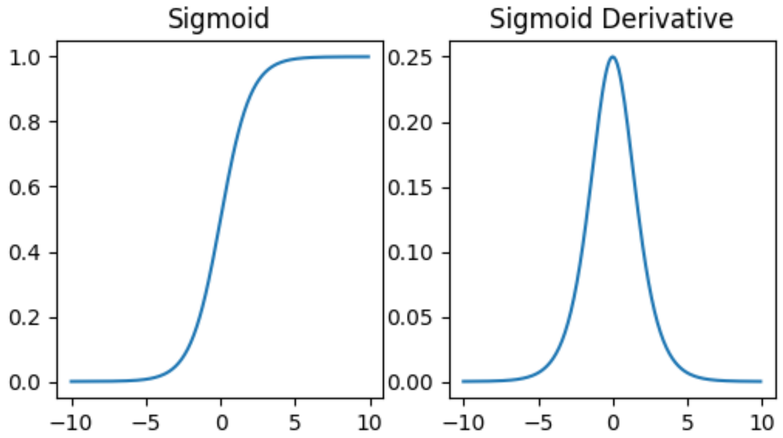
where $x \in \mathbb{R}^3, \hat{y} \in \mathbb{R}^2, W_1 \in \mathbb{R}^{4\times 3}$, $W_2 \in \mathbb{R}^{2\times 4}, b_1 \in \mathbb{R}^4,$ and $b_2 \in \mathbb{R}^2$

https://en.wikipedia.org/wiki/Artificial_neural_network

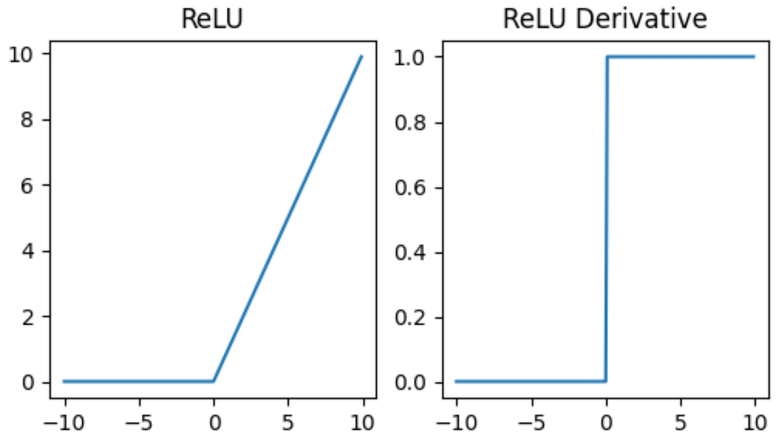
# Common activation functions

## Sigmoid

$$\frac{1}{1+e^{-x}}$$



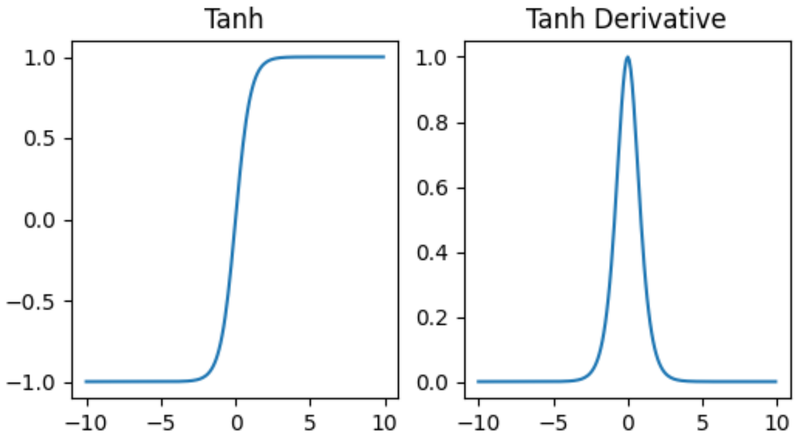

## ReLU



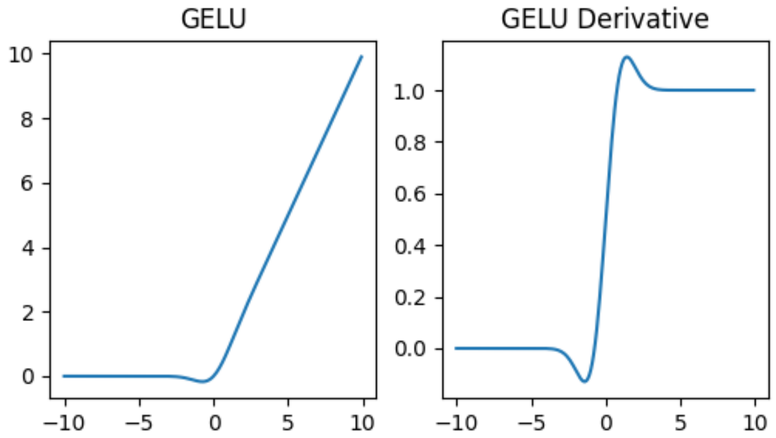

$$\max(0, x)$$

## Tanh

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$




## GeLU




$$\frac{1}{2}x\left(1+\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

# Learning

**Required:**

- Training data $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$
- Model family: some specified function (e.g., $\hat{y} = W_2\sigma(W_1 x + b_1) + b_2$)
  - Number/size of hidden layers, activation function, etc. are FIXED here
- (Differentiable) Loss function $L(y, \hat{y}) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$

**Learning Problem:**

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, \hat{y}^{(i)} = f_\theta(x^{(i)}))$$

# Common loss functions

- **Regression problems:**
  - Euclidean Distance/Mean Squared Error/L2 loss:

    $$L_2(y, \hat{y}) = ||y - \hat{y}||_2^2 = \frac{1}{2}\sum_{i=1}^{k}(y_i - \hat{y}_i)^2$$

  - Mean Absolute Error/L1 loss: $L_1(y, \hat{y}) = ||y - \hat{y}||_1 = \sum_{i=1}^{k}|y_i - \hat{y}_i|$
- **2-way classification:**
  - Binary Cross Entropy Loss: $L_{\text{BCE}}(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$
- **Multi-class classification:** *(for example, words…)*
  - Cross Entropy Loss**:**

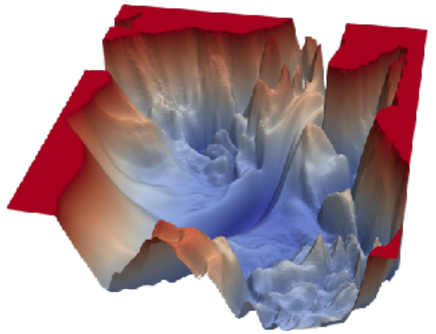
    *(Very related to perplexity!)*

    $$L_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

# Gradient Descent

**Learning Problem:**

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, \hat{y}^{(i)} = f_\theta(x^{(i)}))$$



"Loss landscape" - loss w.r.t $\theta$

https://www.cs.umd.edu/~tomg/projects/landscapes/

- However, finding the global minimum is often impossible in practice (need to search over all of $\mathbb{R}^{\dim(\theta)}$!)
- Instead, get a local minimum with *gradient descent*

**Gradient Descent**

- Learning rate $\alpha \in \mathbb{R}, \alpha > 0$ (often quite small e.g., 3e-4)
- Randomly initialize $\theta^{(0)}$
- Iteratively get better estimate with:

Next estimate — Learning rate (step size)

$$\theta^{(i+1)} = \theta^{(i)} - \alpha * \boxed{\frac{\partial L}{\partial \theta}(\theta^{(i)})}$$

Previous Estimate

**Gradient is:**

- the vector of partial derivatives of the parameters with respect to the loss function
- A linear approximation of the loss function at $\theta^{(i)}$

$$\frac{\partial L}{\partial \theta}(\theta^{(i)}) = \begin{bmatrix} \frac{\partial L}{\partial \theta_1^{(i)}} \\ \frac{\partial L}{\partial \theta_2^{(i)}} \\ \vdots \\ \frac{\partial L}{\partial \theta_n^{(i)}} \end{bmatrix}$$

# Stochastic gradient descent

**Gradient Descent:**

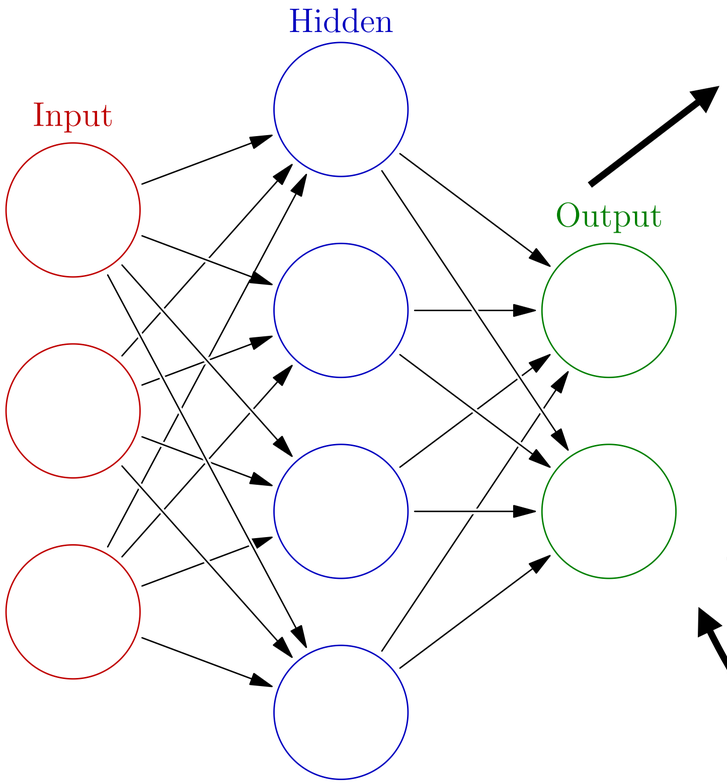
$$\theta^{(i+1)} = \theta^{(i)} - \alpha * \frac{\partial L}{\partial \theta}(\theta^{(i)})$$

- Problem: calculating the true gradient can be very expensive (requires running model on entire dataset!)
- Solution: **Stochastic Gradient Descent**
  - Sample a subset of the data of fixed size (batch size)
  - Take the gradient with respect to that subset
  - Take a step in that direction; repeat
- Not only is it more computationally efficient, but it often finds better minima than vanilla gradient descent
  - Why? Possibly because it does a better job skipping past plateaus in loss landscape

# Backpropagation

One efficient way to calculate the gradient is with **backpropagation**.

Leverages the **Chain Rule**: $\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$




1. Forward Pass

$$h_1 = W_1 x + b_2$$

$$h_2 = \sigma(h_1)$$

$$\hat{y} = W_2 h_2 + b_2$$

2. Calculate Loss

$$L(y, \hat{y})$$

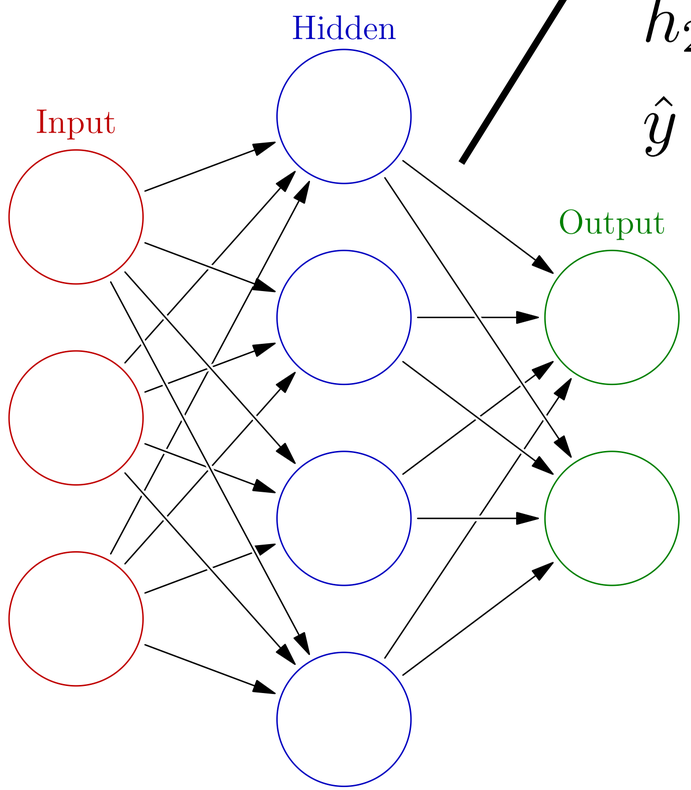
3. Backwards Pass

Calculate the gradient of the loss w.r.t. each parameter using the chain rule and intermediate outputs

# Backpropagation

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

(Long, messy exact derivation below)




## 1. Forward Pass

$$h_1 = W_1 x + b_2$$
$$h_2 = \sigma(h_1)$$
$$\hat{y} = W_2 h_2 + b_2$$

## 2. Calculate Loss

$$L(y, \hat{y})$$

## 3. Backwards Pass

$$\delta_{\hat{y}} := \frac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial (W_2 h_2 + b_2)}{\partial W_2} = \delta_{\hat{y}} \cdot h_2^T$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial (W_2 h_2 + b_2)}{\partial b_2} = \delta_{\hat{y}}$$

$$\delta_{h_2} := \frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial (W_2 h_2 + b_2)}{\partial h_2} = \delta_{\hat{y}} \cdot W_2^T$$

$$\delta_{h_1} := \frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} = \delta_{h_2} \cdot \frac{\partial \sigma(h_1)}{\partial h_1}$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial W_1} = \delta_{h_1} \cdot \frac{\partial (W_1 x + b)}{\partial W_1} = \delta_{h_1} \cdot x^T$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial b_1} = \delta_{h_1} \cdot \frac{\partial (W_1 x + b)}{\partial b_1} = \delta_{h_1}$$

# Classification with deep learning

- For classification problems (like next word-prediction…) we want to predict a **probability distribution** over the label space
- However, neural networks' output $y \in \mathbb{R}^d$ is not guaranteed (or likely) to be a probability distribution
- To force the output to be a probability distribution, we apply the **softmax function**

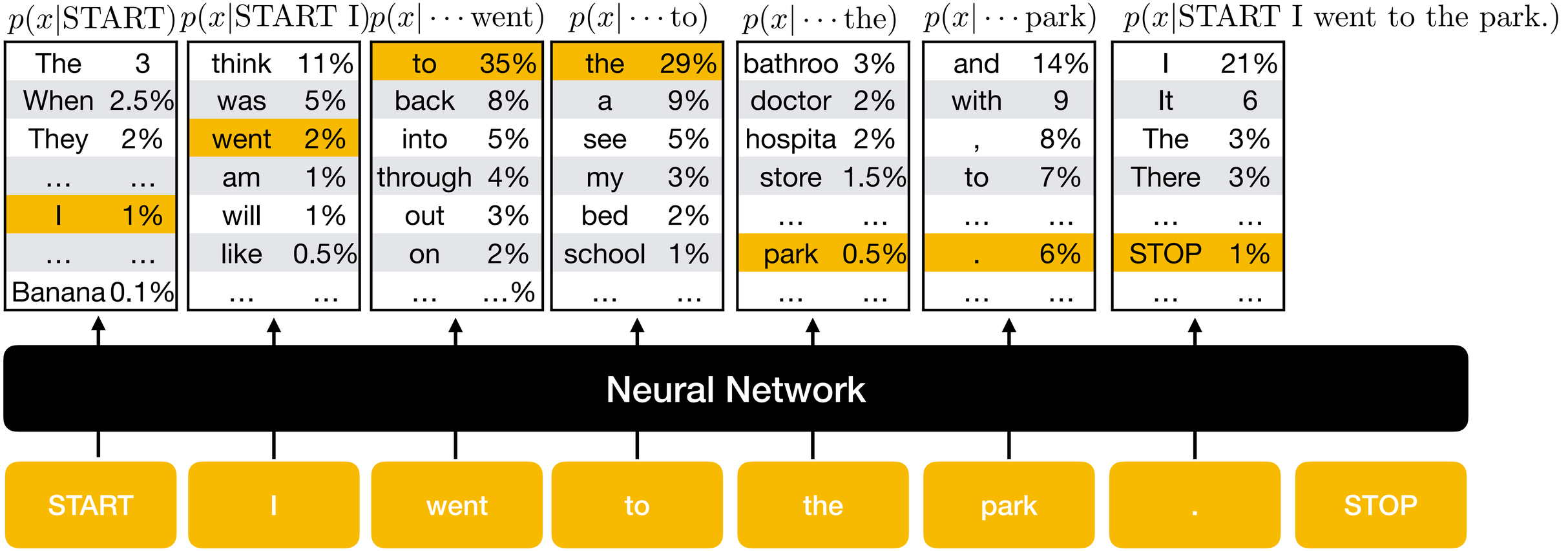
$$\mathrm{softmax}(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{d} \exp(y_j)}$$

  - The values $y$ before applying the softmax are often called "logits"

# Language modeling with neural networks: RNNs

# Inputs/Outputs

- **Input:** sequences of words (or tokens)
- **Output:** probability distribution over the next word (token)



# Neural language models

*But neural networks take in real-valued vectors, not words…*

- Use one-hot or learned embeddings to map from words to vectors!
  - Learned embeddings become part of parameters $\theta$

*Neural networks output vectors, not probability distributions…*

- Apply the softmax to the outputs!
- What should the size of our output distribution be?
  - Same size as our vocabulary $|\mathcal{V}|$

*Don't neural networks need a fixed-size vector as input? And isn't text variable length?*

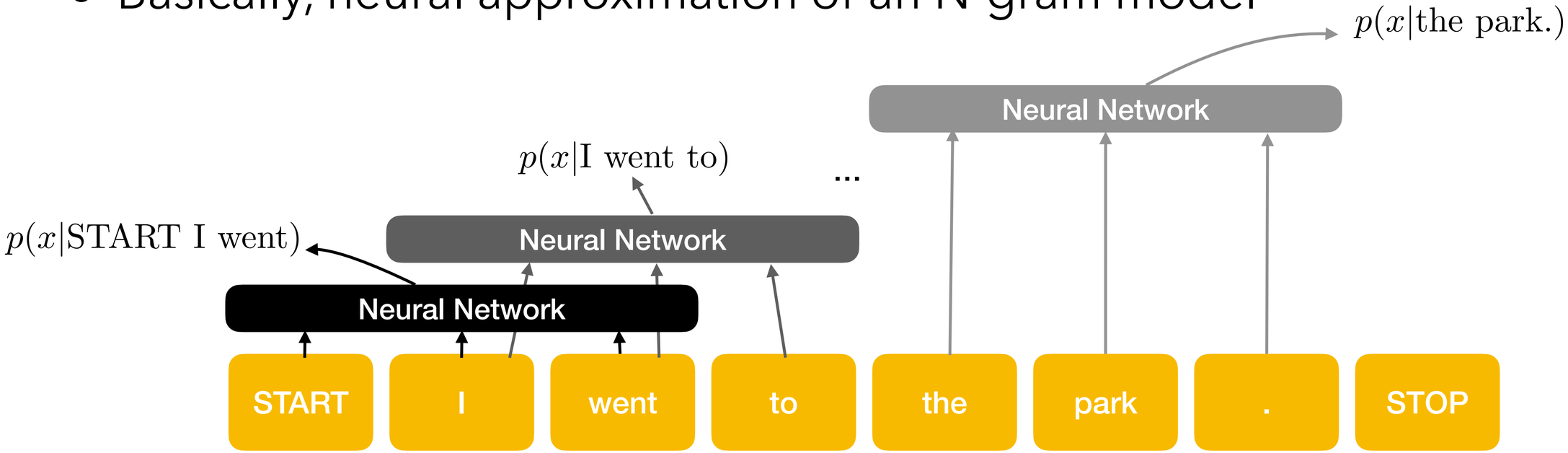
- Ideas?

# Sliding window

*Don't neural networks need a fixed-size vector as input? And isn't text variable length?*

**Idea 1: Sliding window of size N**

- Cannot look more than N words back
- Basically, neural approximation of an N-gram model

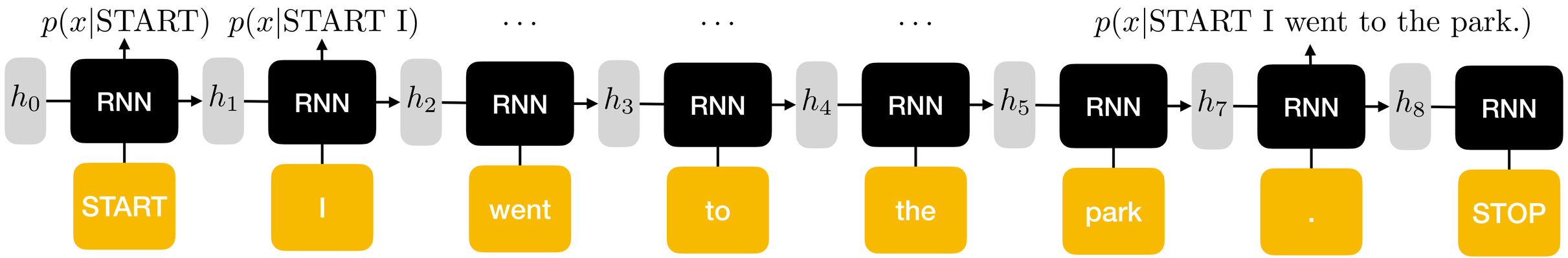


# Recurrent neural networks

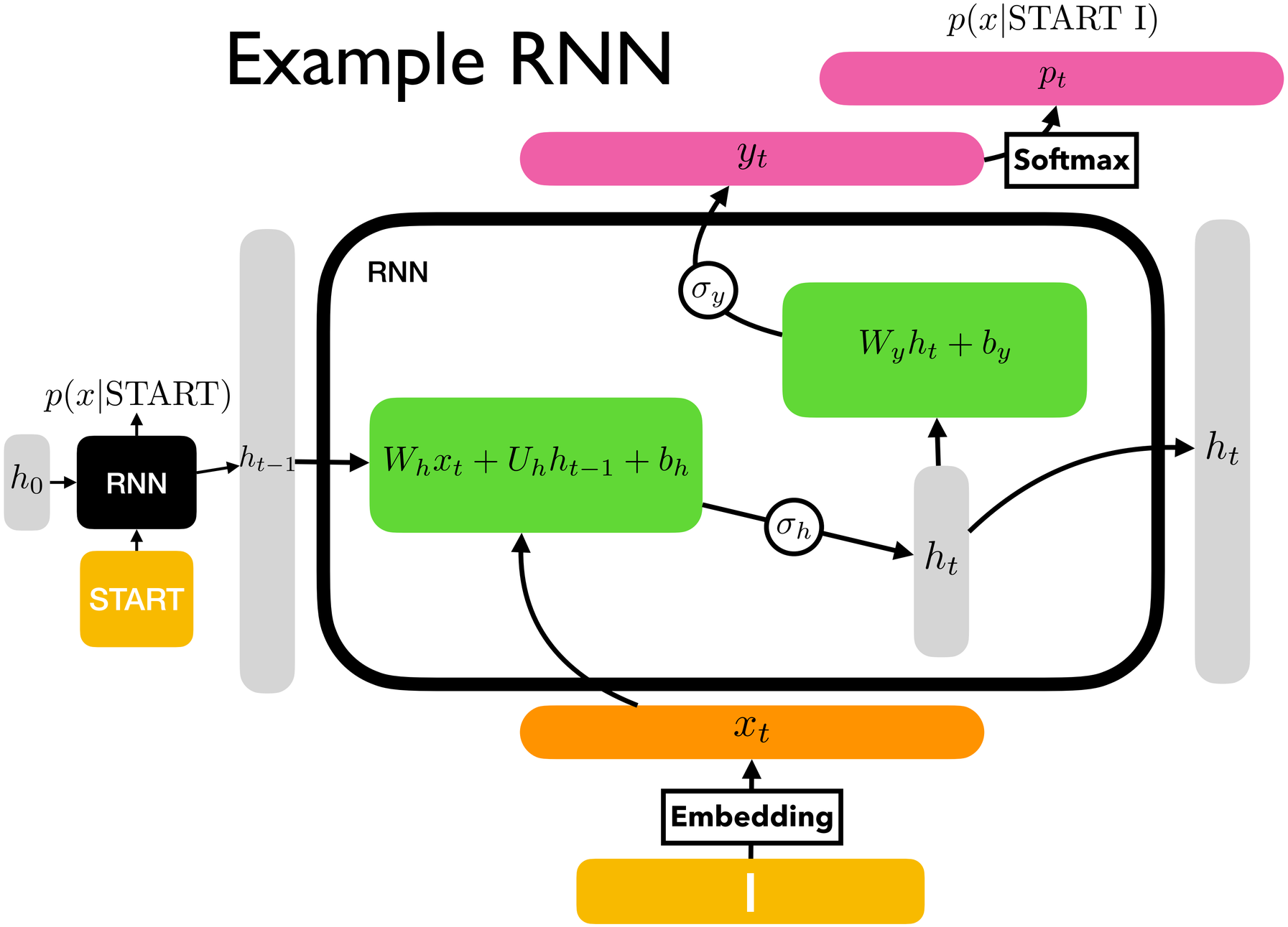
**Idea 2: Recurrent Neural Networks (RNNs)**

Essential components:

- One network is applied recursively to the sequence
- *Inputs:* previous hidden state $h_{t-1}$, observation $x_t$
- *Outputs:* next hidden state $h_t$, (optionally) output $y_t$
- Memory about history is passed through hidden states



# Example RNN




**Variables:**

$x_t$: input (embedding) vector

$y_t$: output vector (logits)

$p_t$: probability over tokens

$h_{t-1}$: previous hidden vector

$h_t$: next hidden vector

$\sigma_h$: activation function for hidden state

$\sigma_y$: output activation function

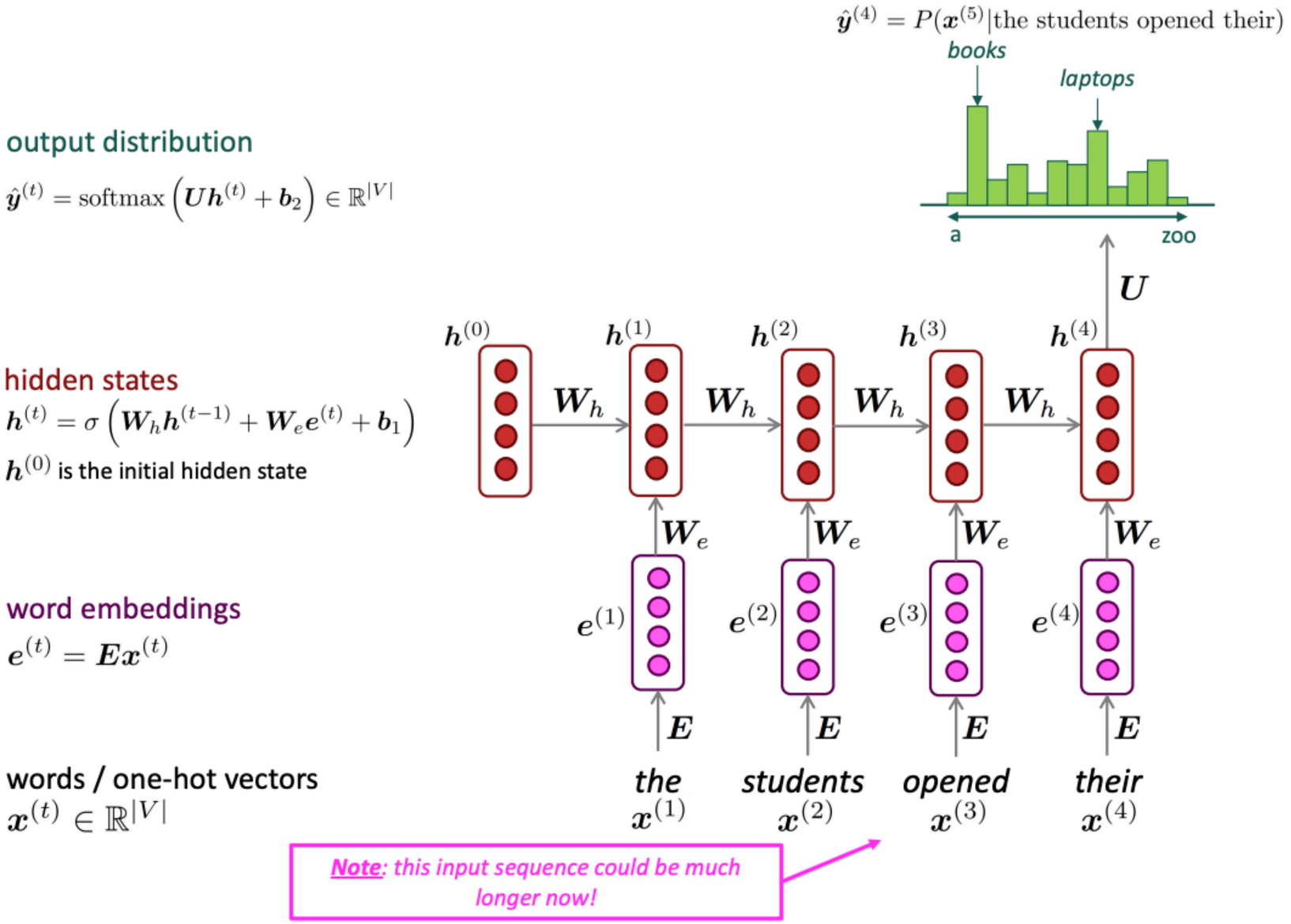
**Equations:**

$$h_t := \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$

$$y_t := \sigma_y(W_y h_t + b_y)$$

$$p_{t_i} = \frac{\exp(y_{t_i})}{\sum_{i=j}^{d} \exp(y_{t_j})}$$

# Example RNN


output distribution
$\hat{y}^{(t)} = \text{softmax}\left(Uh^{(t)} + b_2\right) \in \mathbb{R}^{|V|}$
hidden states
$h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right)$
$h^{(0)}$ is the initial hidden state
word embeddings
$e^{(t)} = Ex^{(t)}$
words / one-hot vectors
$x^{(t)} \in \mathbb{R}^{|V|}$
$\hat{y}^{(4)} = P(x^{(5)}|\text{the students opened their})$
books
laptops
a
zoo
$U$
$W_h$
$W_e$
$E$
$h^{(0)}$
$h^{(1)}$
$h^{(2)}$
$h^{(3)}$
$h^{(4)}$
$e^{(1)}$
$e^{(2)}$
$e^{(3)}$
$e^{(4)}$
the
students
opened
their
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
$x^{(4)}$
Note: this input sequence could be much longer now!

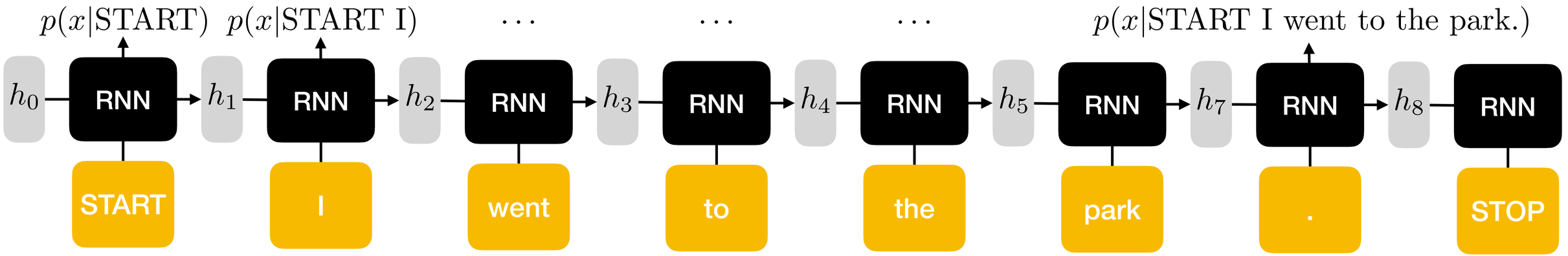
# Recurrent neural networks

- How can information from time an earlier state (e.g., time 0) pass to a later state (time t?)
  - Through the hidden states!
  - Even though they are continuous vectors, can represent very rich information (up to the entire history from the beginning)

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \times P(w_2 \mid w_1) \times P(w_3 \mid w_1, w_2) \times \ldots \times P(w_n \mid w_1, w_2, \ldots, w_{n-1})$$

$$= P(w_1 \mid \mathbf{h}_0) \times P(w_2 \mid \mathbf{h}_1) \times P(w_3 \mid \mathbf{h}_2) \times \ldots \times P(w_n \mid \mathbf{h}_{n-1})$$

No Markov assumption here!



# Training procedure

E.g., if you wanted to train on "<START>I went to the park.<STOP>"…

1. Input/Output Pairs

$\mathcal{D}$

| x (input) | y (output) |
|---|---|
| START | I |
| START I | went |
| START I went | to |
| START I went to | the |
| START I went to the | park |
| START I went to the park | . |
| START I went to the park. | STOP |

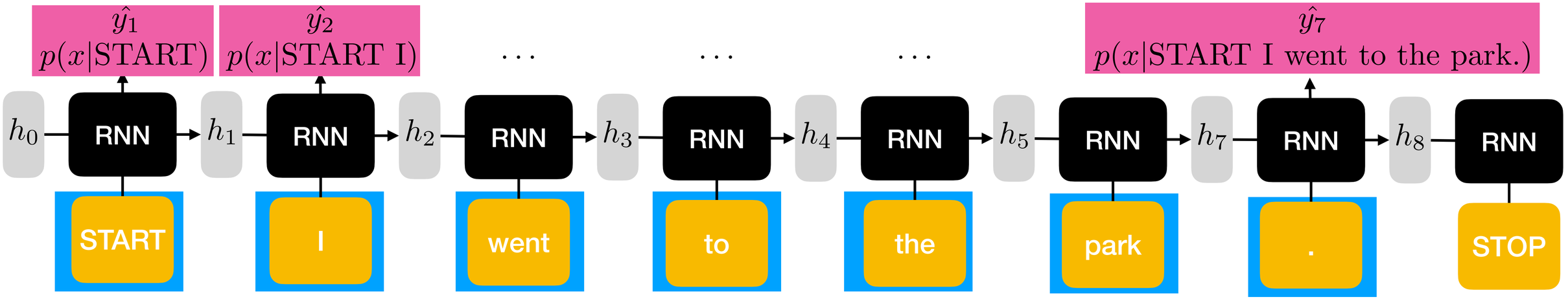
# Training procedure

1. Input/Output Pairs

$\mathcal{D}$

| x (input) | y (output) |
|---|---|
| START | I |
| START I | went |
| START I went | to |
| START I went to | the |
| START I went to the | park |
| START I went to the park | . |
| START I went to the park. | STOP |

2. Run model on (batch of) $x$'s from data $\mathcal{D}$ to get probability distributions $\hat{y}$ (running softmax at end to ensure valid probability distribution)

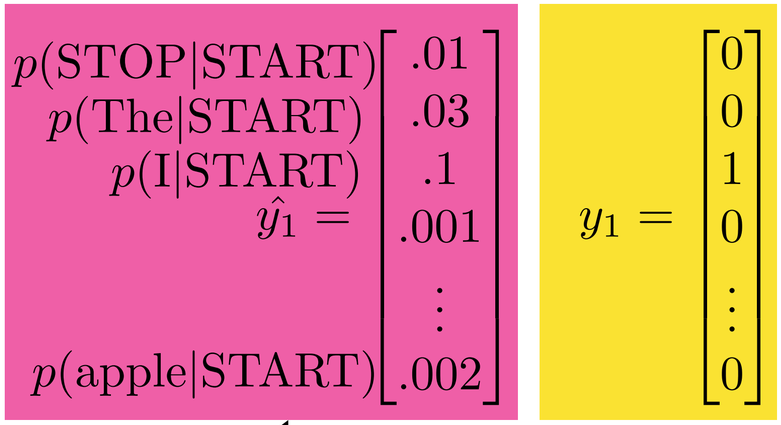


# Training procedure

2. Run model on (batch of) $x$'s from data $\mathcal{D}$ to get probability distributions $\hat{y}$

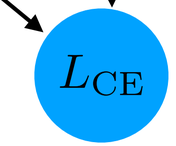
3. Calculate loss compared to true $y$'s (Cross Entropy Loss)

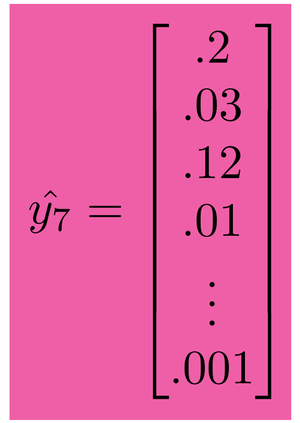
$$L_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

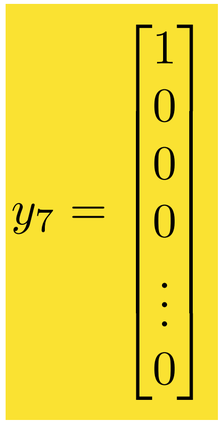


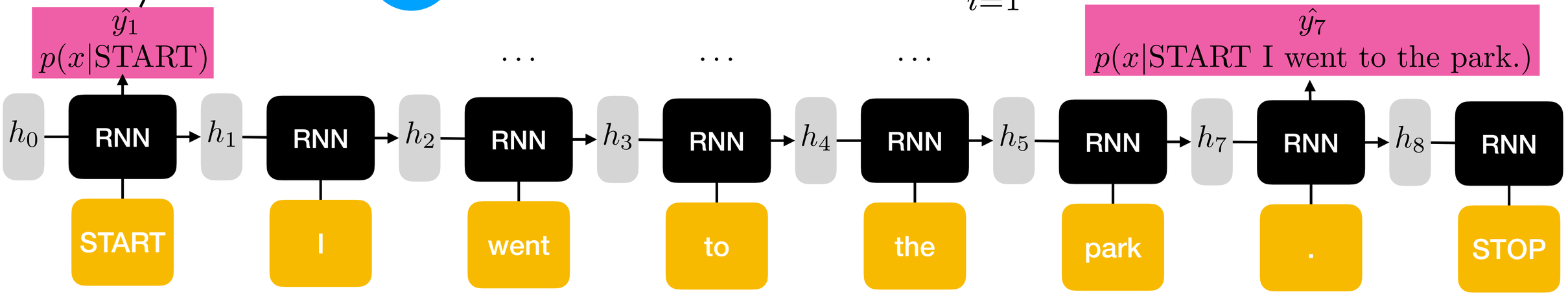






# Training procedure

$$L_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

$$\hat{y}_1 = \begin{bmatrix} p(\text{STOP}|\text{START}) \\ p(\text{The}|\text{START}) \\ p(\text{I}|\text{START}) \\ \vdots \\ p(\text{apple}|\text{START}) \end{bmatrix} \begin{bmatrix} .01 \\ .03 \\ .1 \\ .001 \\ \vdots \\ .002 \end{bmatrix}$$

$$y_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

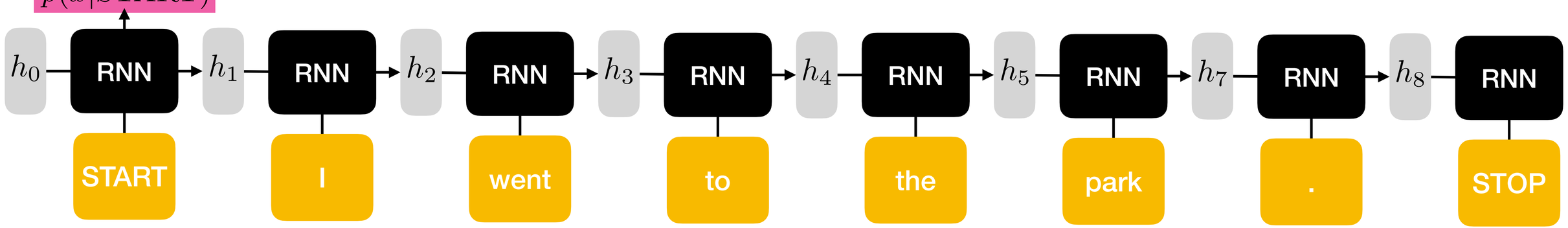
(Actual observed word)

3. Calculate loss compared to true $y$'s (Cross Entropy Loss)

$L_{\text{CE}}$

$$L_{\text{CE}}(y_1, \hat{y}_1) = -0 * \log(.01) - 0 * \log(.03) - 1 * -\log(.1) - \cdots - 0 * \log(.002)$$
$$= -\log(.1) = -\log(p(\text{I}|\text{START}))$$

$\hat{y}_1$
$p(x|\text{START})$



# Training procedure - gradient descent step

1. Get training x-y pairs from batch
2. Run model to get probability distributions over $\hat{y}$
3. Calculate loss compared to true $y$
4. Backpropagate to get the gradient
5. Take a step of gradient descent

$$\theta^{(i+1)} = \theta^{(i)} - \alpha * \frac{\partial L}{\partial \theta}(\theta^{(i)})$$