Deep Generative Models

Chapter 1: Text Generation via Language Models

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Summer 2025

Generating Text

Let's start by a basic task:

we intend to build a model that generates text

+ *Why you think it is basic?*
- Well! Text has some *nice properties*

A text coming from a natural language has two key properties

1. *It has semantics*
   ↳ *"The honey reads a duck" does not make sense!*
2. *It has syntax*
   ↳ *"You am perfect" is grammatically wrong!*

This is thus easier to learn how to write a meaningful and correct text

Generating Text

We intend to build a model that generates text

+ *Recall me! What do you mean by a model?*
- Most of the time, *we mean a NN!*

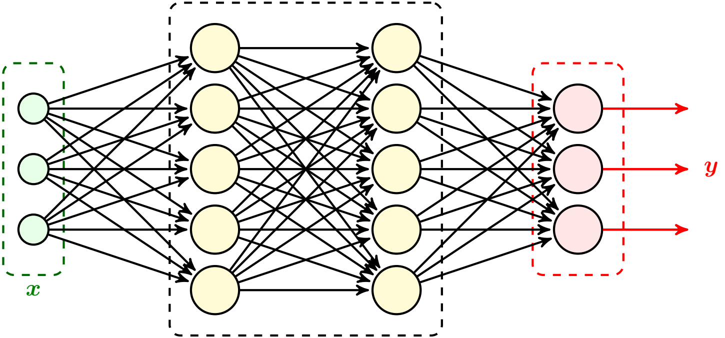
Recap: Learning Model

A learning model is a parameterized mapping $f_{\mathbf{w}} : \mathbb{X} \mapsto \mathbb{V}$ whose parameters $\mathbf{w}$ can be freely tuned $\equiv$ learned

Recap: Training a Model

Training refers to the procedure of finding the right choice of $\mathbf{w}$ from the dataset $\mathbb{D}$ such that $f_{\mathbf{w}}$ can capture the pattern in the data

Recap: *Simple Learning Model*



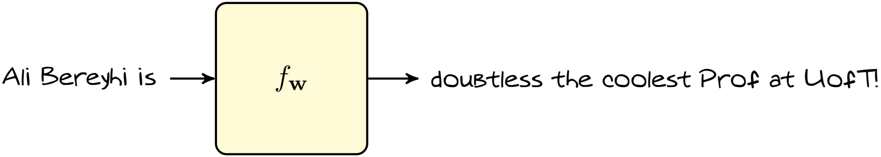
Here, $f_{\mathbf{w}}$ is the function realized by the NN

- *It relates $\boldsymbol{x}$ to $\boldsymbol{y}$*
- *We can learn its weights $\mathbf{w}$*

Text Generating Model

Putting it in simple words: *we aim to train a NN which*

- *starts with an incomplete English text, and*
- *generates the next words till it's complete*



Text Generating Machine: *Requirements*

+ *What kind of property this model should have?*
- It should generate coherent text
  - ↳ *It should have right sentiment*
    - ✔ *The kid plays with ball*
    - ✖ *Water eats the duck*
  - ↳ *It should be consistent with the language syntax*
    - ✔ *She is writing down a text*
    - ✖ *She am write poem*

This text-generating model is what we call a language model

Language Model

Language Model *(Informal)*

A language model is a learning model which can start from an incomplete text and complete it to a coherent text

In this definition there are a few points undefined!

- *What is a text? How can we turn a text to a mathematical object?*
- *How can we precisely define the coherence?!*

Language Model: *Formulation*

We can concretely define a text by noting that

- *A text is a sequence of characters*
- *There are a finite number of characters in any language*
  - ↳ *A basic English text takes characters among 95 possible choices*
- *We can give each character an index $i = 0 : I - 1$*
  - ↳ *I is typically called vocabulary size*

character	*index*
a	0
A	1
⋮	⋮
(	93
)	94

Language Model: *Formulation*

A text can then be represented as a *sequence of character indices*

$$x_1, x_2, \dots, x_T : x_t \in \{1, \dots, I\}$$

We are usually more convenient with one-hot representation

$$\textit{Character } t \textit{ is } \mathbf{A} \equiv x_t = 1 \rightsquigarrow \boldsymbol{x}_t = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

A text can then be represented as a sequence of one-hot vectors

$$\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_T : \mathbf{1}_{x_t} \in \{0, 1\}^I$$

Language Model: *Tokenization*

This is the most simple form of *tokenization*

$$\textit{each } x_t \leftrightsquigarrow \mathbf{1}_{x_t} \textit{ is a token index}$$

In general tokens can be words, sub-words, or characters

Tokenization

The procedure of breaking a text corpus into smaller units called tokens

1. *Break whole text into small tokens*
2. *Give each token an index and save it in a vocabulary*

GPT-4 and GPT-4o have roughly 100K and 200K tokens!

token	*index*
the	0
ing	1
⋮	⋮

Language Model: *Tokenization*

+ *Tokenization seems to be a time-consuming task!*
- Sure! We need to go through the whole corpus

Corpus

Corpus is the body of text we use to train our model $\approx$ *text dataset*

There are various algorithms for tokenization

- *Character-Level*
  ↳ *What we did in our example*
- *Word-Level*
  ↳ *Set each token to be a word in the sentence*
- *Byte Pair Encoding (BPE)*
  ↳ *Iteratively builds a vocabulary of most frequent tokens*
- ...

Language Model: *Embedding*

After tokenization, we have our text as *a sequence*

$$\mathbf{1}_{x_1}, \mathbf{1}_{x_2}, \ldots, \mathbf{1}_{x_T} : \mathbf{1}_{x_t} \in \{0,1\}^I$$

where I is the vocabulary size

+ *Does it mean that in GPT-4o each $\mathbf{1}_{x_t}$ is of size 200K?!*
- *Yes! And, I agree! It sounds too much!*
+ *But, do we really need to add a dimension for every single token?*
- *No! This is why we do embedding*

Language Model: *Embedding*

To understand embedding, let's first *denote vocabulary as a* `<dictionary>`

$$\mathbb{V} = \{\texttt{token} : x \;\; \textit{for} \;\; x = 0 : I - 1\}$$

Embedding

Embedding represents each index $x \in \mathbb{V}$ by a fixed-size vector $\boldsymbol{x} \in \mathbb{R}^E$

Mathematically, embedding is a linear transform: *let* $\mathbf{E} \in \mathbb{R}^{E \times I}$

$$\textit{embedding of } x \equiv \texttt{embd}\left(x\right) = \mathbf{E}\mathbf{1}_x = \begin{array}{c} \cdots \quad x \quad \cdots \\ \downarrow \\ \left[\ldots \quad \boldsymbol{x} \quad \ldots\right] \end{array} \begin{array}{cc} \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \end{bmatrix} & \begin{array}{c} \vdots \\ \\ \leftarrow x \\ \vdots \end{array} \end{array}$$

Embedding: *Example*

Say my corpus is "the good, the bad and the ugly"

Let's build the vocabulary

token	*index*
the	0
good	1
,	2
bad	3
and	4
ugly	5

This means that

$$\textit{and} \equiv 4 \leftrightsquigarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

and the corpus can be represented by indices as

$$0, 1, 2, 0, 3, 4, 0, 5$$

Embedding: *Example*

Say my corpus is "the good, the bad and the ugly"

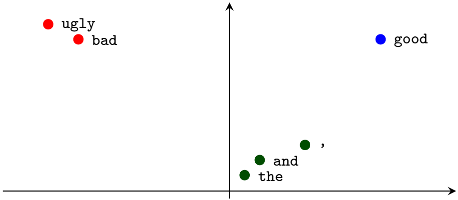
Let's build the vocabulary

token	*index*
the	0
good	1
,	2
bad	3
and	4
ugly	5

We embed them with 2D embedding vectors

$$\mathbf{E} = \begin{bmatrix} 0.1 & 1 & 0.5 & -1 & 0.2 & -1.2 \\ 0.1 & 1 & 0.3 & 1 & 0.2 & 1.1 \end{bmatrix}$$

which we can visualize as



Embedding: *Few Notes*

The embedding matrix $\mathbf{E}$ is learned in practice

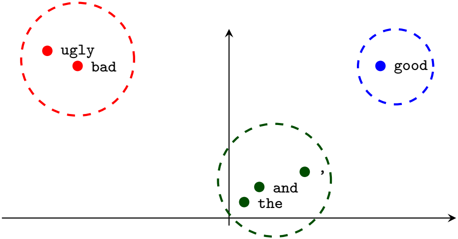
- *We initiate $\mathbf{E}$ with some initial EI weights*
- *In each training iteration we update $\mathbf{E}$*

$$\mathbf{E} \leftarrow \mathbf{E} - \eta \nabla_{\mathbf{E}} \hat{R}$$

learning rate

Empirical Risk $\equiv$ Loss

By the end of training, embeddings are semantically grouped



Back to Language Model

Language Model *(Informal)*

A language model is a learning model which can start from an incomplete text and complete it to a coherent text

We can now make it a bit more formal

Language Model *(pseudo-formal)*

A language model starts from a sequence of tokens

$$x_1, \ldots, x_T : x_t \in [0 : I - 1]$$

and completes it to a coherent sequence

$$x_1, \ldots, x_T, x_{T+1}, \ldots, x_{T+L}$$

Language Distribution

+ *How can we define the coherence?!*
- Well! A coherent sentence is *the one said most likely by a native*

We can assume that a each text completion happens with a probability

- *coherent completions happen with high probabilities*
- *semantically or syntactically wrong completions occur with low probability*

Example: *Say we are to complete the sentence "It is great to hear that"*

$$\Pr\{\textit{you have succeeded} | \textit{It is great to hear that}\} = 0.015$$
$$\Pr\{\textit{potato is round} | \textit{It is great to hear that}\} = 0.0001$$
$$\Pr\{\textit{potato ate the chicken} | \textit{It is great to hear that}\} = 10^{-6}$$
$$\Pr\{\textit{potato am good} | \textit{It is great to hear that}\} = 10^{-9}$$
$$\Pr\{\textit{Kartoffel bin gut} | \textit{It is great to hear that}\} = 10^{-15}$$

Language Distribution

(Conditional) Language Distribution

Given a sequence of tokens $\boldsymbol{x}_1, \dots, \boldsymbol{x}_T$, the language distribution describes the probability of the following tokens, i.e.,

$$p\left(x_{T+1}, \dots, x_{T+L} \middle| x_1, \dots, x_T\right)$$

A coherent text is the one drawn from the language distribution

- *With high probability it is a semantically and syntactically correct sentence*
  ↳ *In practice we always see such samples!*
- *With very low probability it contains semantic or grammatical mistakes*
  ↳ *In practice we never see such samples!*

Language Model: *Definition*

Language Model: *Definition*

A language model starts from a sequence of tokens

$$x_1, \dots, x_T : x_t \in [0 : I - 1]$$

and completes it to

$$x_1, \dots, x_T, x_{T+1}, \dots, x_{T+L}$$

by sampling from the language distribution $p\left(x_{T+1}, \dots, x_{T+L} | x_1, \dots, x_T\right)$

Sampling from Distribution

It is important to recall that when we sample independently, we get a typical sequence which means we see most of the time what is likely

Building LM: *Model*

To start: *say we want to generate only a single token*[1]

$$x_1, \ldots, x_T \rightarrow x_{T+1}$$

We try to do this using deep learning

- *We set a model and make a dataset*
- *We train the model on the dataset by minimizing a risk function*

The model in this case is NN which

- *takes token indices $x_1, \ldots, x_T$ as input*
- *computes the probability of the next token at the output*
  - ↳ *It computes $\Pr\left\{x_{T+1} = x | x_1, \ldots, x_T\right\}$ for all x's in vocabulary*
  - ↳ *How can we do it? Just use a Softmax activation at the output layer*

[1] We later extend it to a general text completer

Building LM: *Model*

$$x_1, \dots, x_T \longrightarrow \boxed{f_{\mathbf{w}}} \longrightarrow \mathbf{p}_{t+1}$$

Once $x_1, \dots, x_T$ is fed, the model gives us

$$\mathbf{p}_{T+1} = f_{\mathbf{w}}\left(x_1, \dots, x_T\right) \in [0, 1]^I$$

and we sample from it!

***Example:** In PyTorch*

$x_{T+1} \leftarrow$ `torch.multinomial(`$\mathbf{p}_{T+1}$`, #smpl=1)`

Building LM: *Data*

We typically collect a corpus of data from internet

- *A long set of coherent texts collected from reliable sources like Wikipedia*
- *Sometimes, we have multiple corpora: one text, one programming, etc*
- *Models like GPT-4 are trained on the body of internet!*

Say, we have a single corpus: *we can tokenize it and convert it into*

$$a_1, a_2, \ldots, a_N$$

and N *is huge, e.g.,* $N \approx 3.3$ *billion in BERT!*

+ *How can we make data samples from this corpus?*
- *Let's do it together!*

Building LM: *Making Data Samples*

We make a labeled sample by randomly choosing a sequence of length T

- *randomly choose* $j \in \{1, \dots, N - T\}$
- *set the sample sequence to*

$$\{x_{1:T}\} = a_j, a_{j+1}, \dots, a_{j+T-1}$$

- *label this sequence by token* $v = a_{j+T}$

We repeat this procedure B times to make a batch of labeled samples

$$\mathbb{B} = \{(\{x_{1:T}\}_b, v_b) : b = 1 : B\}$$

we use this batch to compute sample gradients for SGD updates

↳ *in each iteration of SGD, we make an independent* $\mathbb{B}$

↳ *Think about what an epoch means in this case* ☺

Maximum Likelihood Learning

Let's focus on a batch B: *for each sample* $\{x_{1:T}\}_b$ *label* v_b *is a likely output*

↳ v_b *is the next token in a coherent text*

This means that the LM should sample token v_b *with high probability*

Likelihood *(informal)*

The likelihood over batch $\mathbb{B}$ *is*

$$\mathcal{L}(\mathbf{w}) = \prod_{b=1}^{B} \Pr\left(x_{T+1} = v_b \middle| \{x_{1:T}\}_b \text{ goes into } f_{\mathbf{w}}\right)$$
$$= \prod_{b=1}^{B} f_{\mathbf{w}}\left(\{x_{1:T}\}_b\right)[v_b]$$

Maximum Likelihood Learning

We try to maximize the likelihood computed by our model

↳ *this makes the model generate likely next tokens*

This means that we use SGD to solve

$$
\begin{aligned}
\max_{\mathbf{w}} \mathcal{L}(\mathbf{w}) \equiv \max_{\mathbf{w}} \log \mathcal{L}(\mathbf{w}) &= \max_{\mathbf{w}} \sum_{b=1}^{B} \log f_{\mathbf{w}}\left(\{x_{1:T}\}_b\right)[v_b] \\
&\equiv \min_{\mathbf{w}} \sum_{b=1}^{B} -\log f_{\mathbf{w}}\left(\{x_{1:T}\}_b\right)[v_b] \\
&= \min_{\mathbf{w}} \mathtt{average}\left\{\mathrm{CE}\left(\mathbb{B}\right)\right\}
\end{aligned}
$$

Moral of Story

We solve a classification problem by minimizing the average cross-entropy

Simple Assumption: *Order-One Markov Process*

+ *Can you stop talking so much statistical?!* ☹
– Sure! Let's try a simple model ☺

The most simple way of language modeling is the so-called bi-gram model

Bi-Gram Model

In bi-gram model, we assume that the language approximately describes a Markov process of order one, i.e., at time t

$$p\left(x_{t+1} | x_1, \ldots, x_t\right) \approx p\left(x_{t+1} | x_t\right)$$

Thus, to generate the next token, we only need to know the current one

+ *But, this sounds like a bad approximation!*
– Well! It can still give related consecutive words in a text

Basic Bi-Gram Model

Let's build a very simple bi-gram model: *we embed each token with a vector of size I, i.e., the vocabulary size, and activate it via Softmax*

$$p\left(x_{t+1}|x_t\right) \approx f_{\mathbf{E}}\left(x_t\right) = \mathit{Soft}_{\max}\left(\mathbf{E}\mathbf{1}_{x_t}\right)$$

index	*Embedding*
0	$\mathbf{e}_0$
1	$\mathbf{e}_1$
$\vdots$	$\vdots$
I-1	$\mathbf{e}_{I-1}$

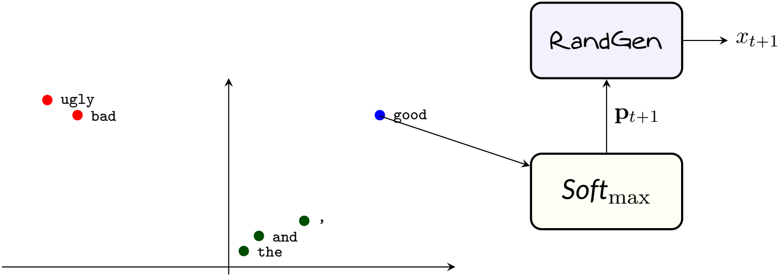
$$\mathbf{E} = \left[\mathbf{e}_0, \ldots, \mathbf{e}_{I-1}\right] \updownarrow I$$

Say $\boxed{x_{1:t}, x_{t+1} = \texttt{6,4,3,1, ?}}$

We predict the next word as

$$p\left(x_{t+1}|\texttt{6,4,3,1}\right) \approx \mathit{Soft}_{\max}\left(\mathbf{e}_1\right) = \begin{cases} p_{t+1,0} \\ p_{t+1,1} \\ \vdots \\ p_{t+1,I-1} \end{cases}$$

Basic Bi-Gram Model



Forward_BiGram(x_t):

1: Set $\boldsymbol{x}_t \leftarrow$ column x_t of $\mathbf{E}$
2: Set $\mathbf{p}_{t+1} \leftarrow$ `softmax`($\boldsymbol{x}_t$)
3: $x_{t+1} \leftarrow$ `multinomial`($\mathbf{p}_{t+1}$, `#smpl=1`)
4: **return** $\mathbf{p}_{t+1}, x_{t+1}$

Training Bi-Gram Model

We can perform *maximum likelihood training* via a *SGD-like optimizer*

`MaxLikeTrain(`$\mathbb{D}$`:train_set):`

for *multiple epochs* **do**
  Sample a batch $\mathbb{B} = \left\{ (x_t, x_{t+1})_b \right\}$ `#` $x_{t,b}, x_{t+1,b} \in [0 : I - 1]$
  for $b = 1 : B$ **do**
    $\mathbf{p}_b, \_ \leftarrow$ `Forward_BiGram`$(x_{t,b})$ `#` $\mathbf{p}_b \in [0, 1]^I$
    Compute $-\nabla_{\mathbf{E}} \log \mathbf{p}_b[x_{t+1,b}]$ `# sample LL grad`
  end for
  Update $\mathbf{E} \leftarrow \mathbf{E} - \eta$`Opt_avg`$\left\{ \nabla_{\mathbf{E}} \mathbf{p}_b \right\}$
end for

Results with Bi-Gram LM

+ *It sounds too basic! How well this model works after training?*
- As you expect!

The Bi-Gram model is extremely limited

✔ *It can complete sentences with a single token missing*

✖ *We cannot use it to generate a long text*

↳ *It only guarantees two consecutive words match*

↳ *I loops over some likely repetitions*

↳ *It returns sentences that are semantically and syntactically wrong*

Reason is Obvious!

Bi-Gram has no memory! It only matches two consecutive words

Adding Memory: *Average Context*

+ *How can we add memory to the model?*
– Well! We need to somehow inject longer text to the model

Let's try a very simple approach: *we modify the bi-gram LM as*

- *We sample longer sequences*
- *To predict token x_{t+1}, we feed the model by sum of all embeddings till t*

$$\boldsymbol{x}_1 \xrightarrow{+} \boldsymbol{x}_2 \xrightarrow{+} \cdots \xrightarrow{+} \boldsymbol{x}_t \xrightarrow{+} \mathbf{c}_t \rightarrow \boxed{\mathrm{Soft}_{\max}} \rightarrow \mathbf{p}_{t+1}$$

Adding Memory: *Average Context*

`ContextAwareLM(`$x_{1:t}$`):`

1: Set context vector to $\mathbf{c}_t \leftarrow \mathbf{0}$
2: **for** $j = 1 : t$ **do**
3: Set $\mathbf{c}_t \leftarrow \mathbf{c}_t +$ column x_j of $\mathbf{E}$
4: **end for**
5: Set $\mathbf{p}_{t+1} \leftarrow$ `softmax(`$\mathbf{c}_t$`)`
6: $x_{t+1} \leftarrow$ `multinomial(`$\mathbf{p}_{t+1}$`, #smpl=1)`
7: **return** $\mathbf{p}_{t+1}, x_{t+1}$

Language Distribution: *More Realistic Markov Process*

If we look from the statistical point of view: *context aware modeling makes a more realistic assumption of the language distribution*

Context Aware Model of Distribution

The context aware model builds the context variable $\mathbf{c}_t$ from the tokens up to x_t

$$\mathbf{c}_t \propto x_1, \ldots, x_t$$

and assumes that

$$p\left(x_{t+1} | x_1, \ldots, x_t\right) \approx p\left(x_{t+1} | \mathbf{c}_t\right)$$

+ *Why is this more realistic?*
– Well! If $\mathbf{c}_t$ maintains all information in $x_{1:t}$ the model is precise!

Training Context Aware Model

It's good to think about the training loop in this case

`MaxLikeTrain(`$\mathbb{D}$`:train_set):`

for *multiple epochs* **do**
    Sample a batch $\mathbb{B} = \left\{ \left(x_{1:T+1} \right)_b \right\}$ `#samples of` T `tokens`
    for $b = 1 : B$ **do**
        for $t = 1 : T$ **do**
            $\mathbf{p}_{t+1,b}, \_ \leftarrow$ `ContextAwareLM(`$x_{1:t,b}$`)` `#`$\mathbf{p}_{t+1,b} \in [0,1]^I$
            Compute $-\nabla_{\mathbf{E}} \log \mathbf{p}_{t+1,b}[x_{t+1,b}]$ `#sample LL grad`
        end for
    end for
    Update $\mathbf{E} \leftarrow \mathbf{E} - \eta$`Opt_avg`$\{\nabla_{\mathbf{E}} \mathbf{p}_{t+1,b}\}$ `#average time and batch`
end for

Results with Context Aware LM

+ *Does this simple modification makes a difference?*
– To some extent Yes, but not too sophisticated

We can readily see that our computation of $\mathbf{c}_t$ *does not capture the whole information in* $x_{1:t}$ *as it has no sequential order*

Sequential Order

$\mathbf{c}_t$ *captures sequential order if it changes by time permutation in* $x_{1:t}$

In our simple modification "Alice drank water" and "water drank Alice" have the same context $\mathbf{c}_t$

$$\mathbf{c}_t = \texttt{embd(Alice)} + \texttt{embd(drank)} + \texttt{embd(water)}$$

So, we can guess that for an acceptable result, we should build better context

Encoding Memory into State: *Recurrence*

We can build more advanced context using *idea of recurrence in RNNs*

Recurrent LM

A recurrent LM consists of an model which

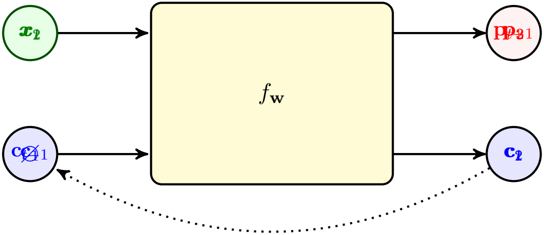
1. *takes embedding $\boldsymbol{x}_t$ and previous context $\mathbf{c}_{t-1}$ as input*
2. *returns a new context $\mathbf{c}_t$ and distribution of next token $\mathbf{p}_{t+1}$ as output*

The context $\mathbf{c}_t$ in this LM is *an encoding of information till token* x_t

- *It is learned by the model; thus, it could be more efficient*
- *It can capture the sequential order*

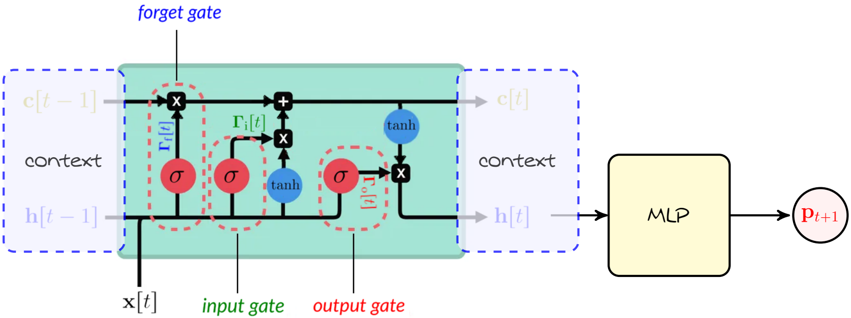
Recurrent LM: *Visualization*

x_1 x_2 $\cdots$ x_{t-1} x_t x_{t+1}



token	*index*	$\mathbf{p}_{t+1}$
the	0	0.1
good	1	0.1
,	2	0.5
bad	3	0.05
and	4	0.2
ugly	5	0.05

Example: *LSTM-based LM*



Training LSTM-based LM

It's good to imagine how we can train this model

`lstmLM_Train(`$\mathbb{D}$`:train_set):`

for *multiple epochs* **do**
  Sample a batch $\mathbb{B} = \left\{ \left(x_{1:T+1} \right)_b \right\}$ `#samples of` T `tokens`
  for $b = 1 : B$ **do**
    Initiate some cell-state $\mathbf{c}$ `#zero context`
    for $t = 1 : T$ **do**
      Set $\boldsymbol{x}_t \leftarrow \texttt{embd} \left(x_t \right)$
      $\mathbf{p}_{t+1,b}, \mathbf{c} \leftarrow \texttt{LSTM}(\boldsymbol{x}_t, \mathbf{c})$ `#`$\mathbf{p}_{t+1,b} \in [0,1]^I$
      Compute $-\nabla \log \mathbf{p}_{t+1,b}[x_{t+1,b}]$ `#sample LL grad`
    end for
  end for
  Update $\mathbf{w} \leftarrow \mathbf{w} - \eta \texttt{Opt\_avg} \left\{ \nabla \mathbf{p}_{t,b} \right\}$ `#average time and batch`
end for

Efficient implementations of this model made first practical LMs[2]

[2] See reading list to see some related further read

Time Delay in Sequential Processing

Recurrent LMs work sequentially, i.e., *when processing a sample of length* T

- *they first compute* $\mathbf{c}_1$ *out of* $\boldsymbol{x}_1$ *and initial zero context*
- *they feed* $\boldsymbol{x}_2$ *with the context* $\mathbf{c}_1$ *to get* $\mathbf{c}_2$
- *they feed* $\boldsymbol{x}_3$ *with updated context* $\mathbf{c}_2$ *to get* $\mathbf{c}_3$
- ...

In other words, they can process a single token at a time

- *This was accepted till 2017 when Transformers were introduced*
- *Transformers provided more efficient way of making context*
  - ↳ *They use self-attention to build context*
  - ↳ *Self-attention enables parallel processing of tokens*

Next Stop: *Transformer-based LMs*

Let's get to transformer-based LMs which describe the current LLMs!

Generation via Recurrent LMs

It's good to think how a recurrent LM generates the whole new text

`lstmLM_Generation(`$x_{1:t}$`:input_text):`

Initiate some cell-state $\mathbf{c}$ `#zero context`
for $j = 1 : t - 1$ **do**
    Set $\boldsymbol{x}_j \leftarrow \texttt{embd}\left(x_j\right)$
    $\_, \mathbf{c} \leftarrow \texttt{LSTM}(\boldsymbol{x}_j, \mathbf{c})$ `#build context`
end for
for $i = 0 : L - 1$ **do**
    $\mathbf{p}_{t+i+1}, \mathbf{c} \leftarrow \texttt{LSTM}(\boldsymbol{x}_{t+i}, \mathbf{c})$ `#next token prob`
    $x_{t+i+1} \leftarrow$ `multinomial(`$\mathbf{p}_{t+i+1}$`, #smpl=1)` `#sample from` $\mathbf{p}_{t+i+1}$
    Set $\boldsymbol{x}_{t+i+1} \leftarrow \texttt{embd}\left(x_{t+i+1}\right)$
end for
return $\{\texttt{token}\left[x_i\right]$ *for* $i = 1 : t + L\}$ `#completed text`