10-423/10-623 Generative AI

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Learning Large Language Models
(Pre-training, fine-tuning, decoding)

Matt Gormley
Lecture 3
Jan. 24, 2024

Reminders

- Homework 0: PyTorch + Weights & Biases
  - Out: Wed, Jan 17
  - Due: Wed, Jan 24 at 11:59pm
  - Two parts:
    1. written part to Gradescope
    2. programming part to Gradescope
  - unique policy for this assignment: we will grant (essentially) any and all extension requests
- Homework 1: Generative Models of Text
  - Out: Thu, Jan 25
  - Due: Wed, Feb 7 at 11:59pm

Q&A

Q: How will I earn the 5% participation points?

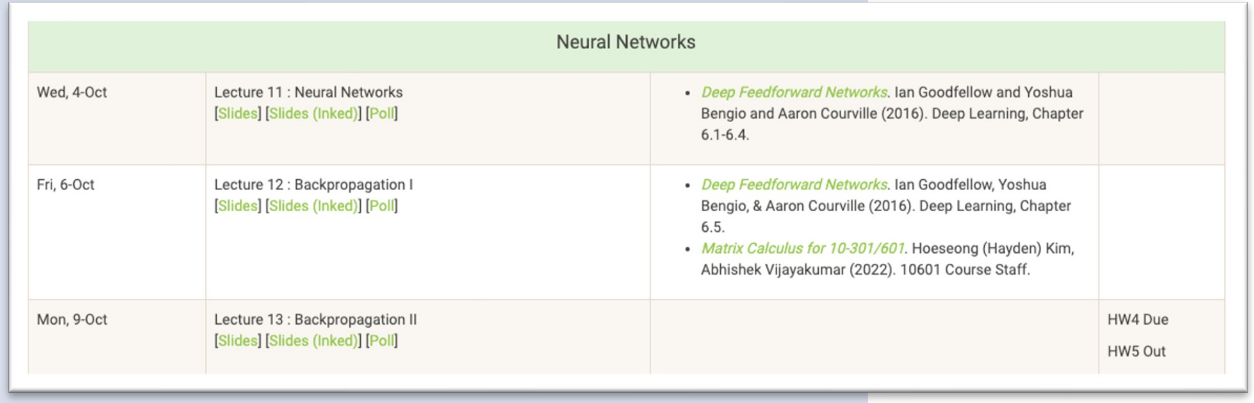
A: Very gradually. There will be a few aspects of the course (polls, surveys, meetings with the course staff) that we will attach participation points to.

Q&A

Q: I'm already feeling a bit lost. The deep learning content is going really fast. What should I do?

A: We are not expecting you to know deep learning already. Consider reviewing the Neural Networks module (Lectures 11 – 13) from 10-301/10-601 Fall 2023.

(Links: Slides and Videos)



Neural Networks

Wed, 4-Oct	Lecture 11 : Neural Networks [Slides] [Slides (Inked)] [Poll]	• *Deep Feedforward Networks*. Ian Goodfellow and Yoshua Bengio and Aaron Courville (2016). Deep Learning, Chapter 6.1-6.4.	
Fri, 6-Oct	Lecture 12 : Backpropagation I [Slides] [Slides (Inked)] [Poll]	• *Deep Feedforward Networks*. Ian Goodfellow, Yoshua Bengio, & Aaron Courville (2016). Deep Learning, Chapter 6.5. • *Matrix Calculus for 10-301/601*. Hoeseong (Hayden) Kim, Abhishek Vijayakumar (2022). 10601 Course Staff.	
Mon, 9-Oct	Lecture 13 : Backpropagation II [Slides] [Slides (Inked)] [Poll]		HW4 Due HW5 Out

Q: But I took 10-301/601 with you and I'm still feeling lost!

A: Uh oh! I must be doing something wrong. Come talk to me and let's figure out together how to fix it.

RECAP

Recall…

Module-based AutoDiff (OOP Version)

Object-Oriented Implementation:

- Let each module be an **object**
- Then allow the **control flow** dictate the creation of the **computation graph**
- No longer need to implement NNBackward(·), just follow the computation graph in **reverse topological order**

```
class Sigmoid(Module)
    method forward($\mathbf{a}$)
        $\mathbf{b} = \sigma(\mathbf{a})$
        return $\mathbf{b}$
    method backward($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
        $\mathbf{g_a} = \mathbf{g_b} \odot \mathbf{b} \odot (1 - \mathbf{b})$
        return $\mathbf{g_a}$
```

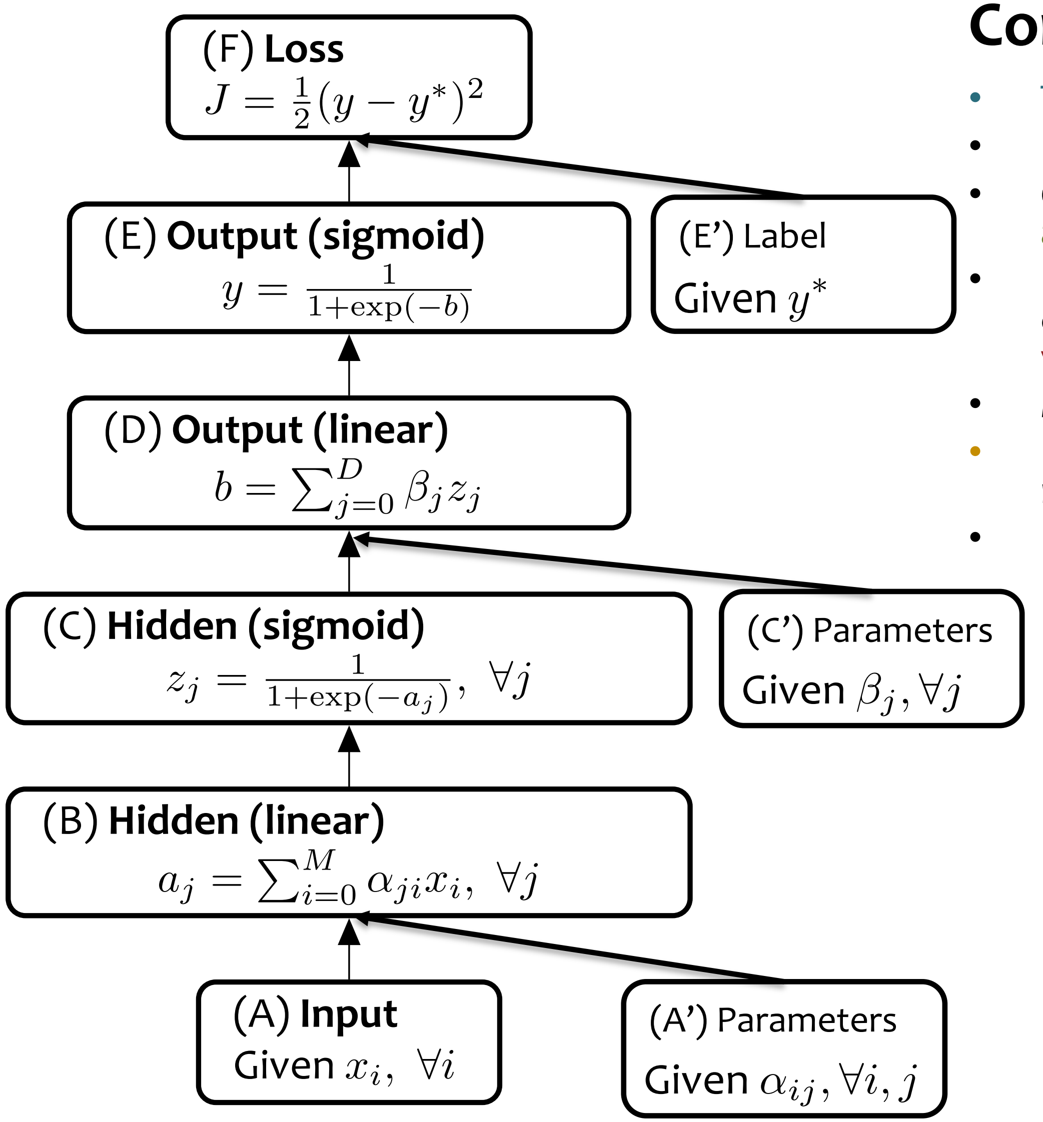
```
class Linear(Module)
    method forward($\mathbf{a}$, $\boldsymbol{\omega}$)
        $\mathbf{b} = \boldsymbol{\omega}\mathbf{a}$
        return $\mathbf{b}$
    method backward($\mathbf{a}$, $\boldsymbol{\omega}$, $\mathbf{b}$, $\mathbf{g_b}$)
        $\mathbf{g}_{\boldsymbol{\omega}} = \mathbf{g_b}\mathbf{a}^T$
        $\mathbf{g_a} = \boldsymbol{\omega}^T\mathbf{g_b}$
        return $\mathbf{g}_{\boldsymbol{\omega}}, \mathbf{g_a}$
```

```
class Softmax(Module)
    method forward($\mathbf{a}$)
        $\mathbf{b} = \texttt{softmax}(\mathbf{a})$
        return $\mathbf{b}$
    method backward($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
        $\mathbf{g_a} = \mathbf{g_b}^T\left(\texttt{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T\right)$
        return $\mathbf{g_a}$
```

```
class CrossEntropy(Module)
    method forward($\mathbf{a}$, $\hat{\mathbf{a}}$)
        $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$
        return $\mathbf{b}$
    method backward($\mathbf{a}$, $\hat{\mathbf{a}}$, $b$, $g_b$)
        $\mathbf{g}_{\hat{\mathbf{a}}} = -g_b(\mathbf{a} \div \hat{\mathbf{a}})$
        return $\mathbf{g_a}$
```

Ways of Drawing Neural Networks

Recall...

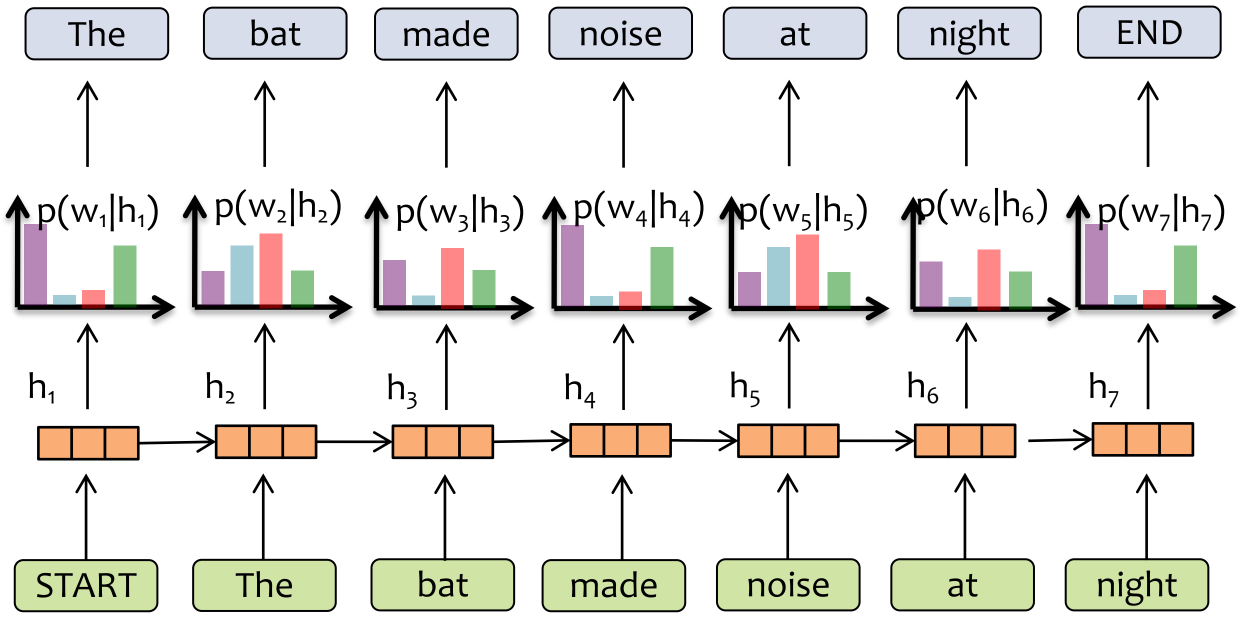


(F) **Loss**
$J = \frac{1}{2}(y - y^*)^2$
(E) **Output (sigmoid)**
$y = \frac{1}{1+\exp(-b)}$
(E') Label
Given y^*
(D) **Output (linear)**
$b = \sum_{j=0}^{D} \beta_j z_j$
(C) **Hidden (sigmoid)**
$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$
(C') Parameters
Given $\beta_j, \forall j$
(B) **Hidden (linear)**
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$
(A) **Input**
Given $x_i, \ \forall i$
(A') Parameters
Given $\alpha_{ij}, \forall i, j$

Computation Graph

- The diagram represents an algorithm
- Nodes are **rectangles**
- One node per intermediate variable in the algorithm
- Node is labeled with the function that it computes (inside the box) and also the variable name (outside the box)
- *Edges are directed*
- Edges do not have labels (since they don't need them)
- For neural networks:
  - Each intercept term should appear as a node (if it's not folded in somewhere)
  - Each parameter should appear as a node
  - Each constant, e.g. a true label or a feature vector should appear in the graph
  - It's perfectly fine to include the loss

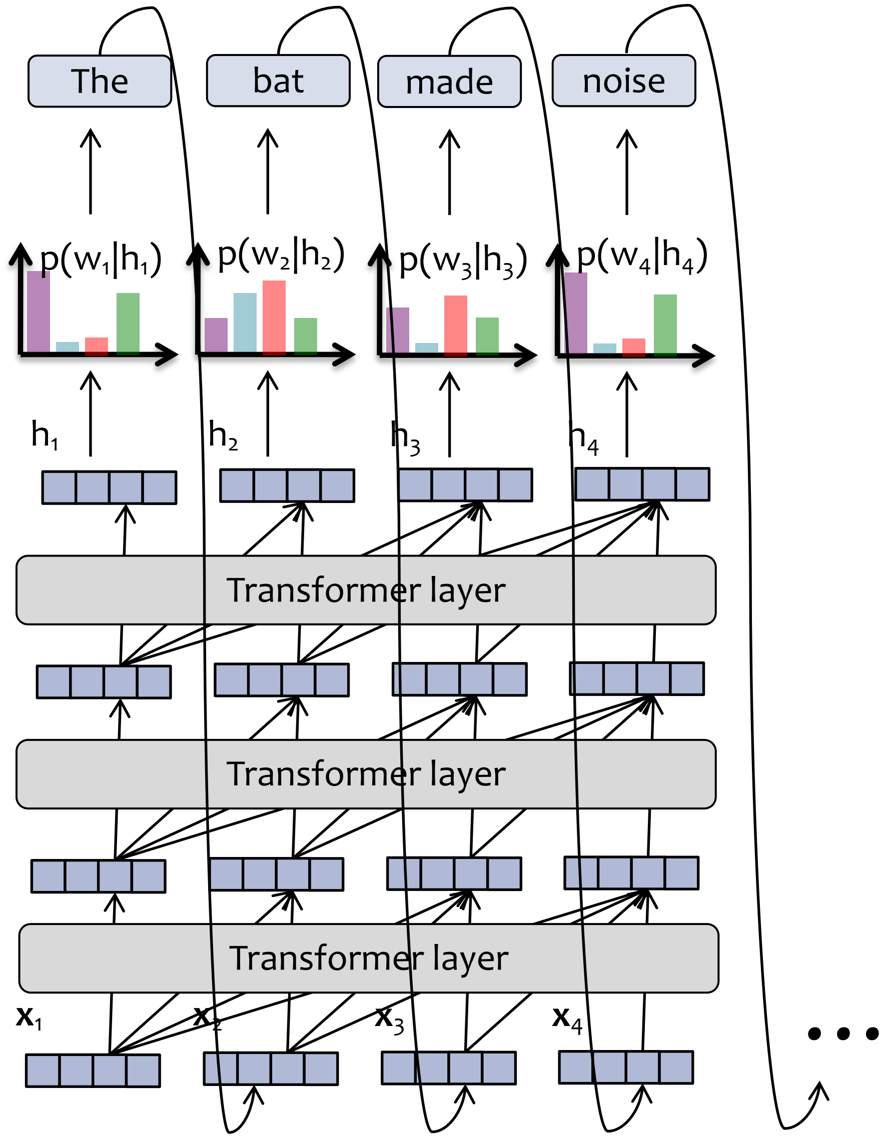
RNN Language Model



Key Idea:
(1) convert all previous words to a **fixed length vector**
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

Transformer Language Model

Recall...



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM.

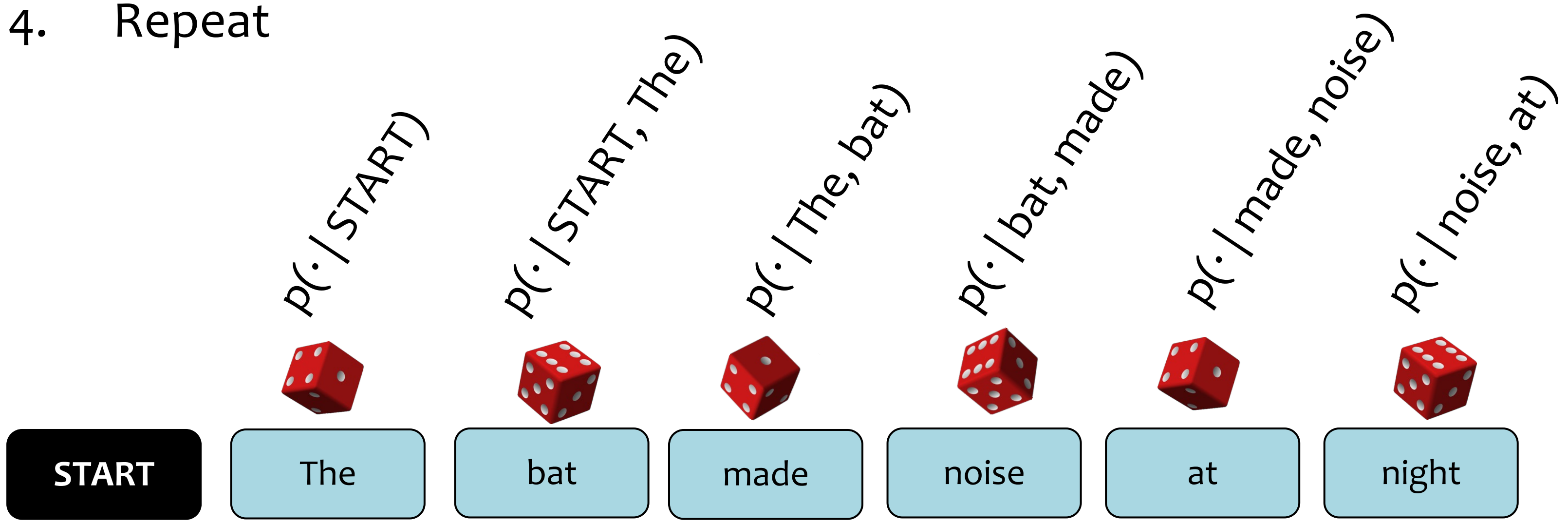
Recall…

Sampling from a Language Model

Question: How do we sample from a Language Model?

Answer:

1. Treat each probability distribution like a (50k-sided) weighted die
2. Pick the die corresponding to $p(w_t \mid w_{t-2}, w_{t-1})$
3. Roll that die and generate whichever word w_t lands face up
4. Repeat



Two parts: Deep Learning and Language Modeling

Recap So Far

Deep Learning

- AutoDiff
  - is a tool for **computing gradients** of a differentiable function, b = f(a)
  - the key building block is a **module** with a forward() and backward()
  - sometimes define f as **code** in forward() by chaining existing modules together
- Computation Graphs
  - are another way to define f (more conducive to slides)
  - so far, we saw two (deep) computation graphs
    - 1) RNN-LM
    - 2) Transformer-LM
    - (Transformer-LM was kind of complicated)

Language Modeling

- key idea: condition on previous words to **sample the next word**
- to define the **probability** of the next word…
  - …n-gram LM uses collection of massive 50k-sided **dice**
  - …RNN-LM or Transformer-LM use a **neural network**
- Learning an LM
  - n-gram LMs are easy to learn: just **count** co-occurrences!
  - so far, we said **nothing about how to learn** an RNN-LM or Transformer-LM
  - So let's figure that out next…

LEARNING A NEURAL NETWORK

Recall…

A Recipe for Machine Learning

1. Given training data:

$$\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$$

2. Choose each of these:
   - Decision function

$$\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

   - Loss function

$$\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R}$$

3. Define goal:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

4. Train with SGD:

(take small steps opposite the gradient)

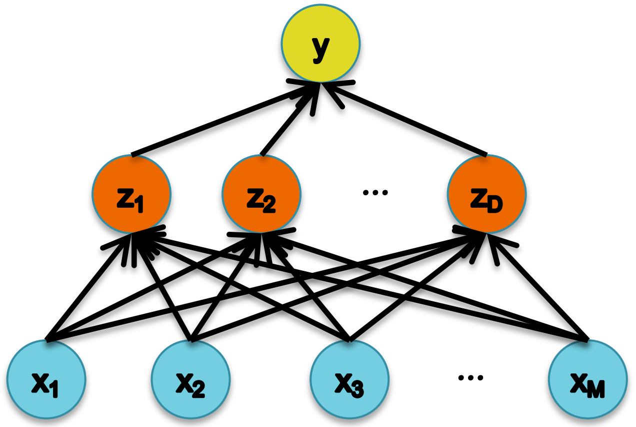
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

Backpropagation

$\frac{\partial J}{\partial b} = g_b$

Recall...

Example:
Neural Network

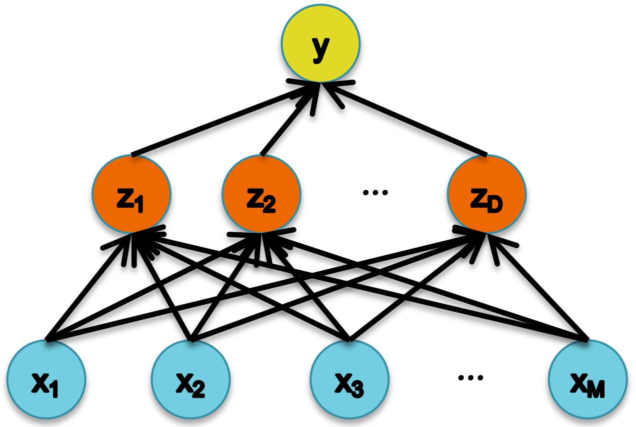


	Forward	Backward
Loss	$\boxed{J} = y^* \log y + (1-y^*)\log(1-y)$	$g_y = \frac{y^*}{y} + \frac{(1-y^*)}{y-1}$
Sigmoid	$y = \frac{1}{1+\exp(-b)}$	$g_b = g_y \frac{\partial y}{\partial b}, \frac{\partial y}{\partial b} = y(1-y)$
Linear	$b = \sum_{j=0}^{D} \beta_j z_j$	$g_{\beta_j} = g_b \frac{\partial b}{\partial \beta_j}, \frac{\partial b}{\partial \beta_j} = z_j$ $g_{z_j} = g_b \frac{\partial b}{\partial z_j}, \frac{\partial b}{\partial z_j} = \beta_j$
Sigmoid	$z_j = \frac{1}{1+\exp(-a_j)}$	$g_{a_j} = g_{z_j} \frac{\partial z_j}{\partial a_j}, \frac{\partial z_j}{\partial a_j} = z_j(1-z_j)$
Linear	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$g_{\alpha_{ji}} = g_{a_j} \frac{\partial a_j}{\partial \alpha_{ji}}, \frac{\partial a_j}{\partial \alpha_{ji}} = x_i$ $g_{x_i} = \sum_{j=0}^{D} g_{a_j} \frac{\partial a_j}{\partial x_i}, \frac{\partial a_j}{\partial x_i} = \alpha_{ji}$

Backpropagation

Recall…

Example:
Neural Network



This whole "Backward" columns is now computed for us automatically by AutoDiff

		Backward
Loss	$y)$	$g_y = \frac{y^*}{y} + \frac{(1-y^*)}{y-1}$
Sigmoid		$g_b = g_y \frac{\partial y}{\partial b}, \frac{\partial y}{\partial b} = y(1-y)$
Linear		$g_{\beta_j} = g_b \frac{\partial b}{\partial \beta_j}, \frac{\partial b}{\partial \beta_j} = z_j$ $g_{z_j} = g_b \frac{\partial b}{\partial z_j}, \frac{\partial b}{\partial z_j} = \beta_j$
Sigmoid	$z_j = \frac{\quad}{1+\exp(-a_j)}$	$g_{a_j} = g_{z_j} \frac{\partial z_j}{\partial a_j}, \frac{\partial z_j}{\partial a_j} = z_j(1-z_j)$
Linear	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$g_{\alpha_{ji}} = g_{a_j} \frac{\partial a_j}{\partial \alpha_{ji}}, \frac{\partial a_j}{\partial \alpha_{ji}} = x_i$ $g_{x_i} = \sum_{j=0}^{D} g_{a_j} \frac{\partial a_j}{\partial x_i}, \frac{\partial a_j}{\partial x_i} = \alpha_{ji}$

Recall...

SGD with Backprop

Example: 1-Hidden Layer Neural Network

Algorithm 1 Stochastic Gradient Descent (SGD)

```
1:  procedure SGD(Training data $\mathcal{D}$, test data $\mathcal{D}_t$)
2:      Initialize parameters $\boxed{\boldsymbol{\alpha}, \boldsymbol{\beta}}$
3:      for $e \in \{1, 2, \ldots, E\}$ do
4:          for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ do
5:              Compute neural network layers:
6:              $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J) = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
7:              Compute gradients via backprop:
8:              $\left.\begin{matrix} \mathbf{g}_{\boldsymbol{\alpha}} = \nabla_{\boldsymbol{\alpha}} J \\ \mathbf{g}_{\boldsymbol{\beta}} = \nabla_{\boldsymbol{\beta}} J \end{matrix}\right\} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$
9:              Update parameters:
10:             $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}}$
11:             $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}}$
12:         Evaluate training mean cross-entropy $\boxed{J_{\mathcal{D}}(\boldsymbol{\alpha}, \boldsymbol{\beta})}$
13:         Evaluate ~~test~~ val. mean cross-entropy $J_{\mathcal{D}_t}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
14:     return parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$
```

SGD and Mini-batch SGD

Recall...

Algorithm 1 SGD

```
1:  Initialize $\theta^{(0)}$
2:
3:
4:  $s = 0$
5:  for $t = 1, 2, \ldots, T$ do
6:      for $i \in$ shuffle$(1, \ldots, N)$ do
7:          Select the next training point $(x_i, y_i)$
8:          Compute the gradient $g^{(s)} = \nabla J_i(\theta^{(s-1)})$
9:          Update parameters $\theta^{(s)} = \theta^{(s-1)} - \eta g^{(s)}$
10:         Increment time step $s = s + 1$
11:     Evaluate average training loss $J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J_i(\theta)$
12: return $\theta^{(s)}$
```

SGD and Mini-batch SGD

Recall...

- reduce variance
- efficiency

Algorithm 1 Mini-Batch SGD

```
1:  Initialize $\theta^{(0)}$ $= \{W_{ah}, W_{ax}, W_{yh}, b_h, b_y\}$
2:  Divide examples $\{1, \dots, N\}$ randomly into batches $\{I_1, \dots, I_B\}$
3:  where $\bigcup_{b=1}^{B} I_b = \{1, \dots, N\}$ and $\bigcap_{b=1}^{B} I_b = \emptyset$
4:  $s = 0$
5:  for $t = 1, 2, \dots, T$ do
6:      for $b = 1, 2, \dots, B$ do
7:          Select the next batch $I_b$, where $m = |I_b|$
8:          Compute the gradient $g^{(s)} = \frac{1}{m} \sum_{i \in I_b} \nabla J_i(\theta^{(s)})$
9:          Update parameters $\theta^{(s)} = \theta^{(s-1)} - \eta g^{(s)}$
10:         Increment time step $s = s + 1$
11:     Evaluate average training loss $J(\theta) = \frac{1}{n} \sum_{i=1}^{n} J_i(\theta)$
12: return $\theta^{(s)}$
```

LEARNING A TRANSFORMER LM

Learning a Language Model

Question: How do we **learn** the probabilities for the n-Gram Model?
Answer: From data! Just **count** n-gram frequencies

…the **cows eat grass**…
…our **cows eat hay** daily…
…factory-farm **cows eat corn**…
…on an organic farm, **cows eat hay** and…
…do your **cows eat grass** or corn?...
…what do **cows eat if** they have…
…**cows eat corn** when there is no…
…which **cows eat which** foods depends…
…if **cows eat grass**…
…when **cows eat corn** their stomachs…
…should we let **cows eat corn**?...

$p(w_t \mid w_{t-2} = \text{cows}, w_{t-1} = \text{eat})$

w_t	$p(\cdot \mid \cdot, \cdot)$
corn	4/11
grass	3/11
hay	2/11
if	1/11
which	1/11

MLE for n-gram LM

- This counting method gives us the **maximum likelihood estimate** of the n-gram LM parameters
- We can derive it in the usual way:
  - **Write the likelihood** of the sentences under the n-gram LM
  - **Set the gradient to zero** and impose the constraint that the probabilities sum-to-one
  - **Solve** for the MLE

Learning a Language Model

MLE for Deep Neural LM

- We can also use maximum likelihood estimation to learn the parameters of an RNN-LM or Transformer-LM too!
- But **not in closed form** – instead we follow a different recipe:
  - Write the negative log-**likelihood** of the sentences under the Deep Neural LM model
  - Compute the **gradient** of the (batch) likelihood w.r.t. the parameters **by AutoDiff**
  - Follow the negative gradient using **Mini-batch SGD** (or your favorite optimizer)

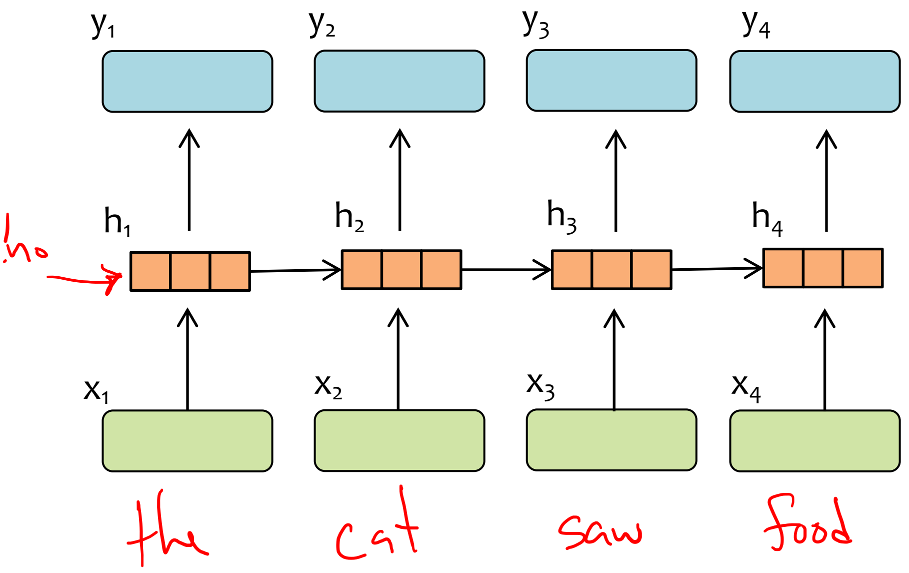
MLE for n-gram LM

- This counting method gives us the **maximum likelihood estimate** of the n-gram LM parameters
- We can derive it in the usual way:
  - **Write the likelihood** of the sentences under the n-gram LM
  - **Set the gradient to zero** and impose the constraint that the probabilities sum-to-one
  - **Solve** for the MLE

RNN

Algorithm 1 Elman RNN

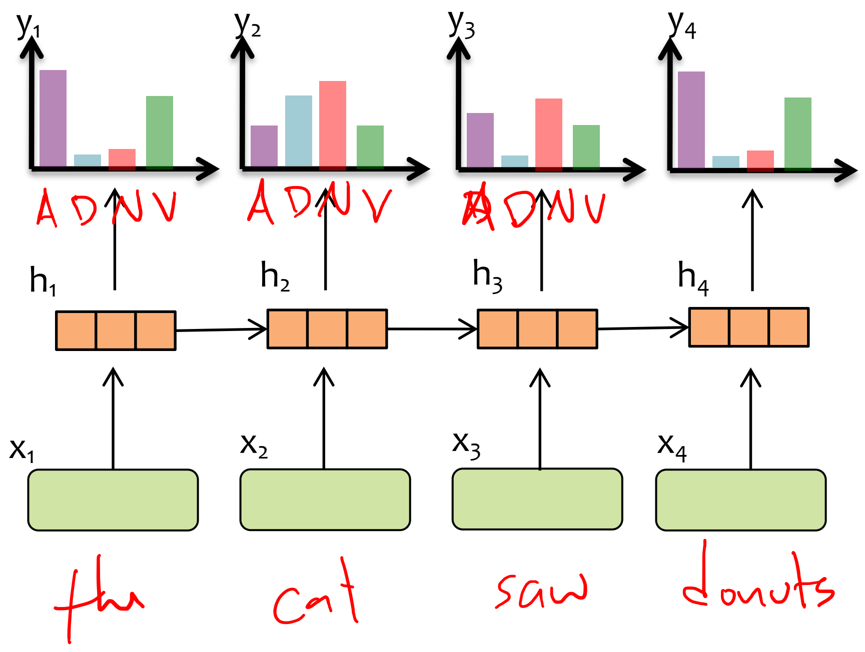
```
1: procedure FORWARD($x_{1:T}, W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
2:     Initialize the hidden state $h_0$ to zeros
3:     for $t$ in $1$ to $T$ do
4:         Receive input data at time step $t$: $x_t$
5:         Compute the hidden state update:
6:             $a_t = \underline{W_{ah}} \cdot \underline{h_{t-1}} + \underline{W_{ax}} \cdot \underline{x_t} + \underline{b_a}$
7:             $h_t = \underline{\sigma}(a_t)$
8:         Compute the output at time step $t$:
9:             $y_t = W_{yh} \cdot h_t + b_y$
```



RNN

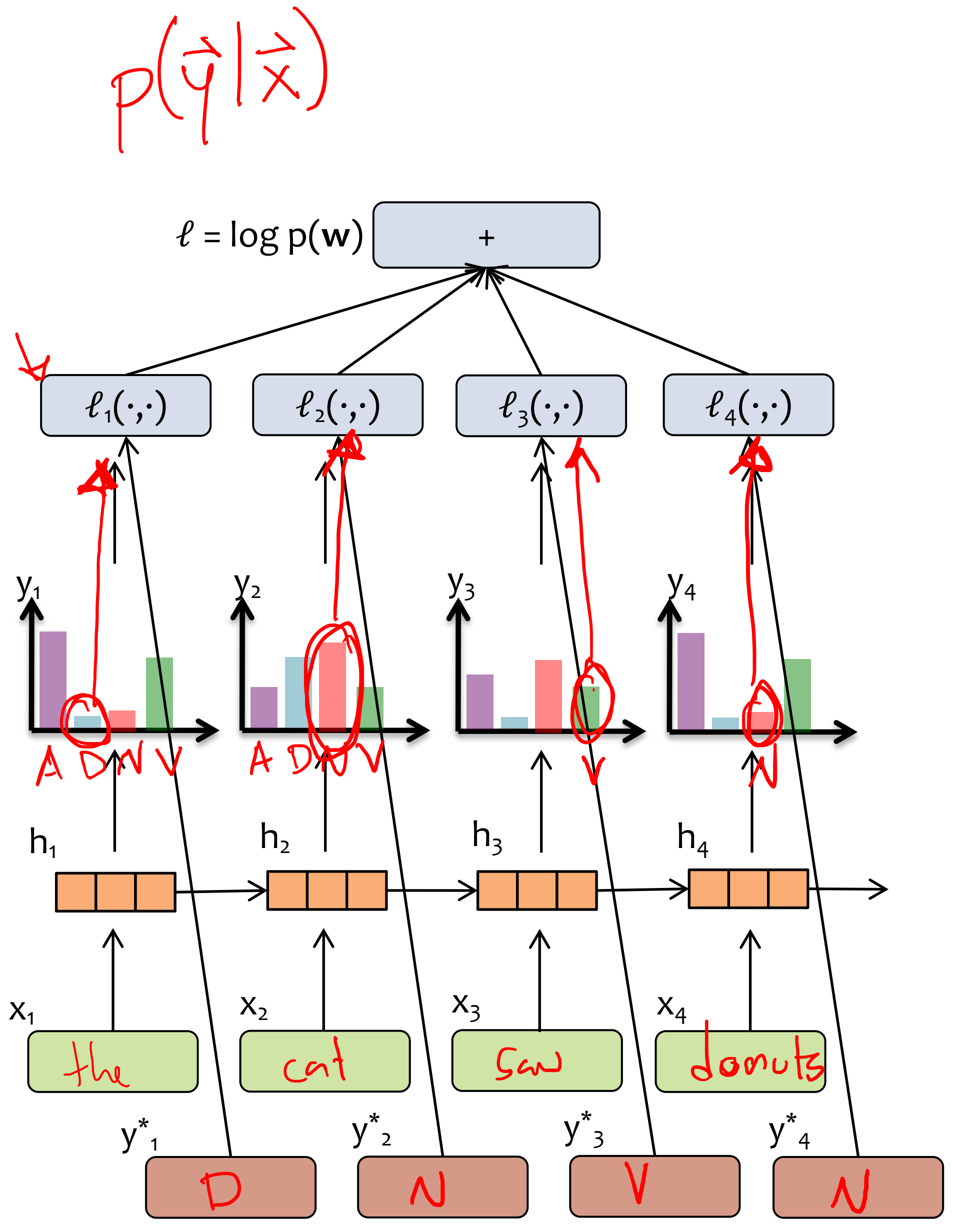
Algorithm 1 Elman RNN

```
1: procedure FORWARD($x_{1:T}, W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
2:     Initialize the hidden state $h_0$ to zeros
3:     for $t$ in $1$ to $T$ do
4:         Receive input data at time step $t$: $x_t$
5:         Compute the hidden state update:
6:             $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
7:             $h_t = \sigma(a_t)$
8:         Compute the output at time step $t$:
9:             $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
```



RNN + Loss

How can we use this to compute the loss for an RNN-LM?



Algorithm 1 Elman RNN + Loss

the gold labels

```
1:  procedure FORWARD($x_{1:T}$, $y^*_{1:T}$ $W_{ah}$, $W_{ax}$, $b_a$, $W_{yh}$, $b_y$)
2:      Initialize the hidden state $h_0$ to zeros
3:      for $t$ in $1$ to $T$ do
4:          Receive input data at time step $t$: $x_t$
5:          Compute the hidden state update:
6:              $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
7:              $h_t = \sigma(a_t)$
8:          Compute the output at time step $t$:
9:              $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
10:         Compute the cross-entropy loss at time step $t$:
11:             $\ell_t = -\sum_{k=1}^{K} (y_t)_k \log((y^*_t)_k)$
12:     Compute the total loss:
13:         $\ell = \sum_{t=1}^{T} \ell_t$
```

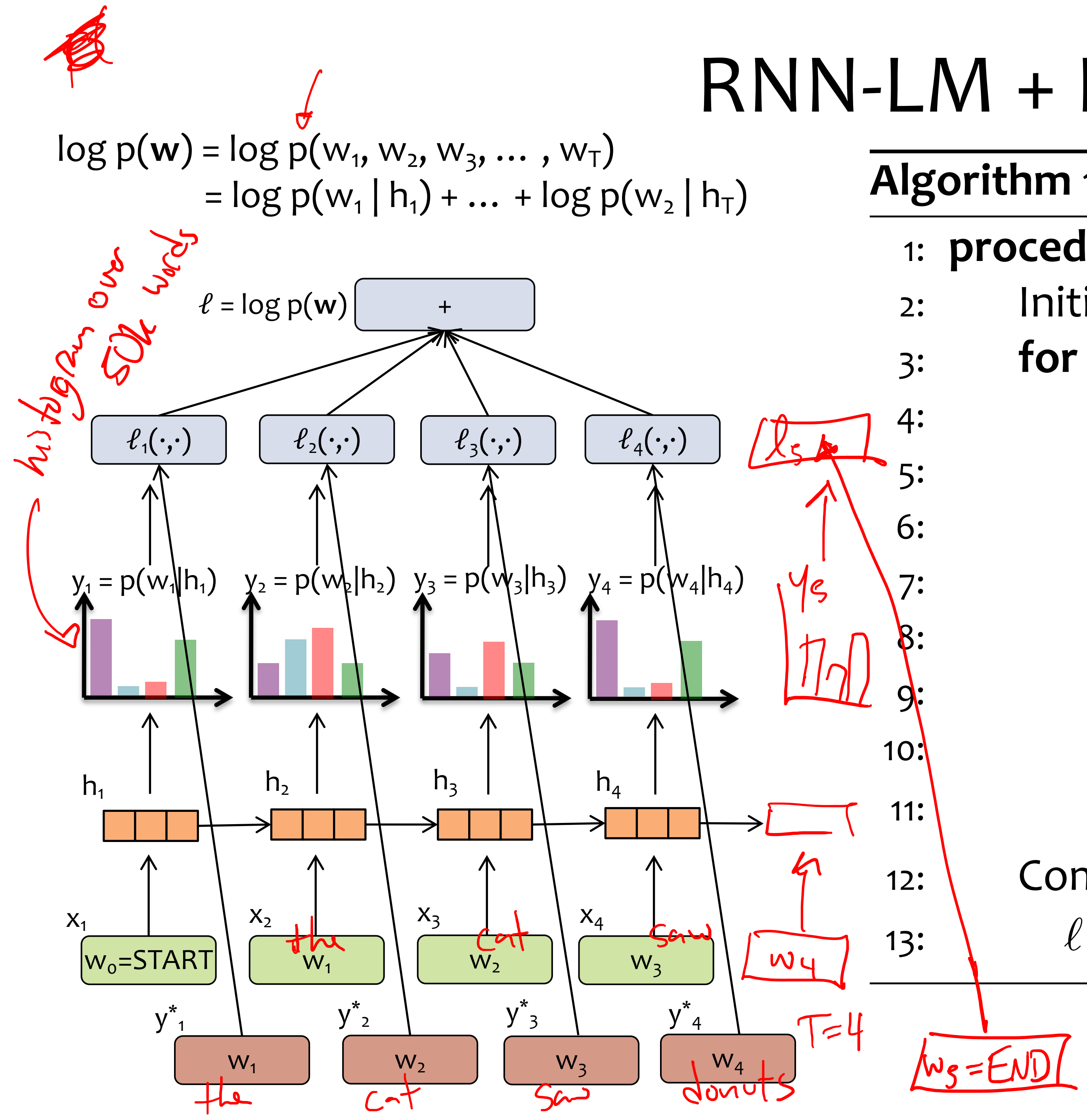
swap these

RNN-LM + Loss

How can we use this to compute the loss for an RNN-LM?

$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \dots, w_T)$

$= \log p(w_1 \mid h_1) + \dots + \log p(w_2 \mid h_T)$



Algorithm 1 Elman RNN + Loss

```
1:  procedure FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
2:      Initialize the hidden state $h_0$ to zeros
3:      for $t$ in $1$ to $T$ do
4:          Receive input data at time step $t$: $x_t$
5:          Compute the hidden state update:
6:              $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
7:              $h_t = \sigma(a_t)$
8:          Compute the output at time step $t$:
9:              $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
10:         Compute the cross-entropy loss at time step $t$:
11:             $\ell_t = -\sum_{k=1}^{K} (y_t)_k \log((y^*_t)_k)$
12:     Compute the total loss:
13:         $\ell = \sum_{t=1}^{T} \ell_t$
```

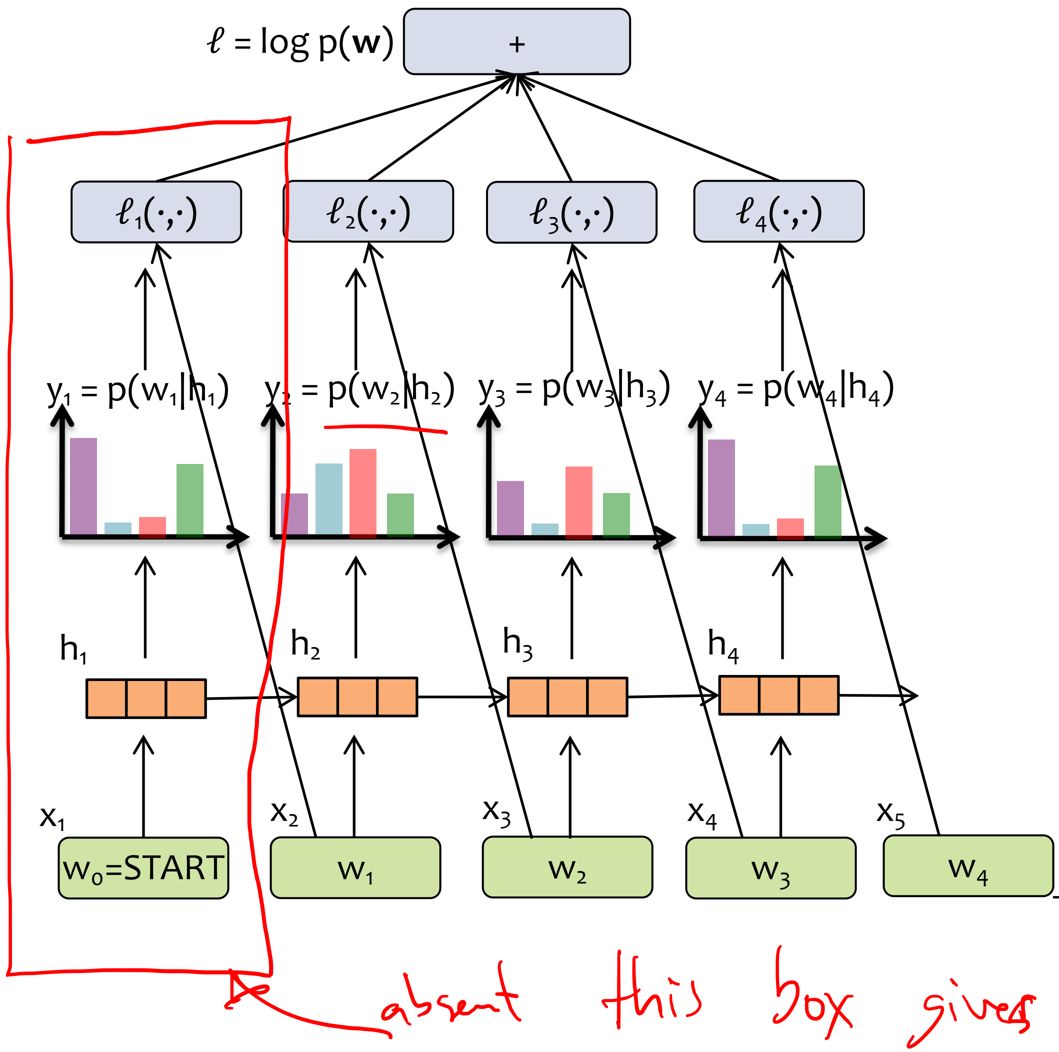
with this we have

$P(w_1, \dots, w_T, w_{T+1} \mid w_0)$

RNN-LM + Loss

How can we use this to compute the loss for an RNN-LM?

$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots, w_T)$$
$$= \log p(w_1 \mid h_1) + \ldots + \log p(w_2 \mid h_T)$$



Algorithm 1 Elman RNN + Loss

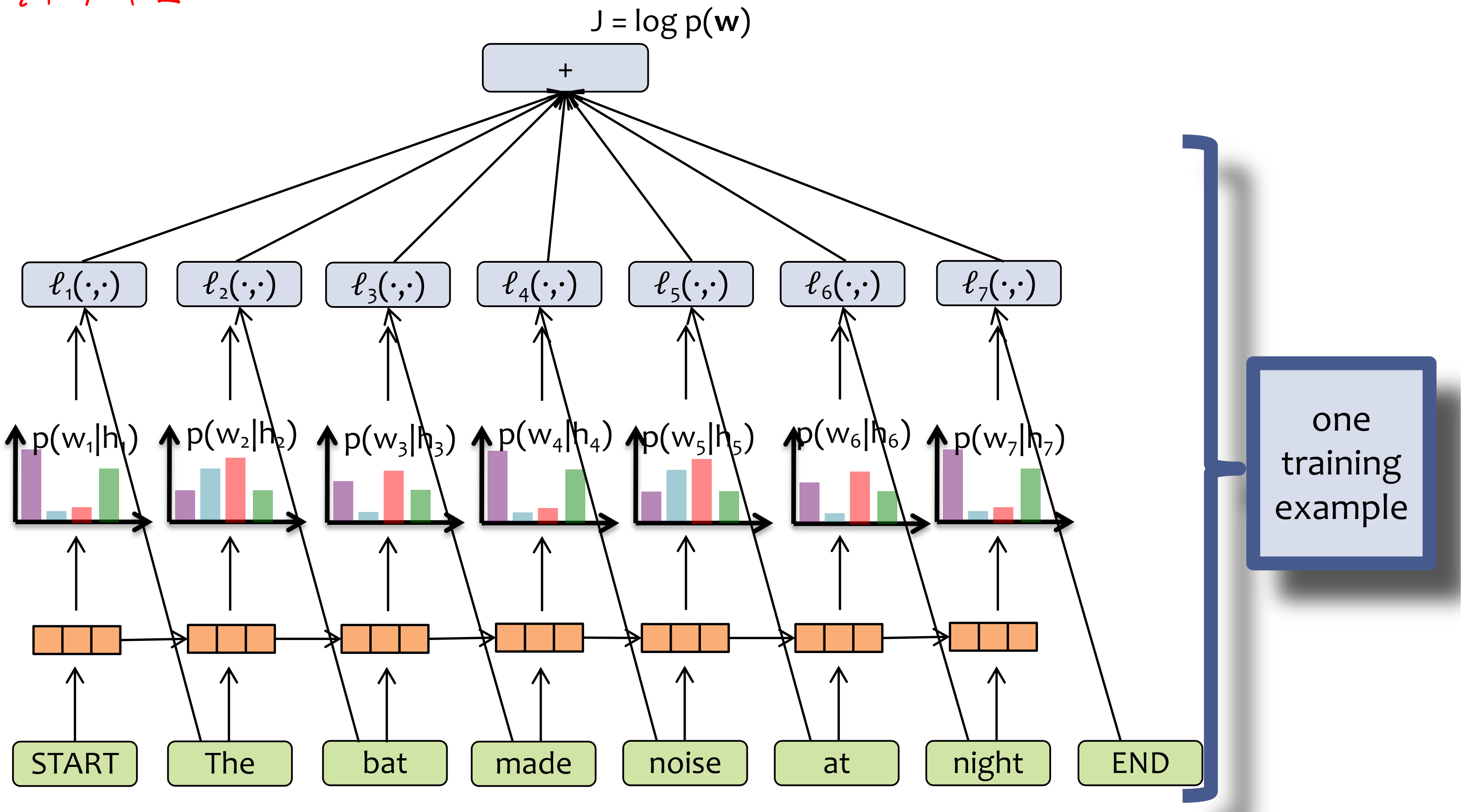
```
1:  procedure FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
2:      Initialize the hidden state $h_0$ to zeros
3:      for $t$ in $1$ to $T$ do
4:          Receive input data at time step $t$: $x_t$
5:          Compute the hidden state update:
6:              $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
7:              $h_t = \sigma(a_t)$
8:          Compute the output at time step $t$:
9:              $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
10:         Compute the cross-entropy loss at time step $t$:
11:             $\ell_t = -\sum_{k=1}^{K} (y_t)_k \log((y^*_t)_k)$
12:     Compute the total loss:
13:         $\ell = \sum_{t=1}^{T} \ell_t$
```

absent this box gives $p(w_2, \ldots, w_T \mid w_1) = p(w_2 \mid h_2)\, p(w_3 \mid h_3)\, p(w_4 \mid h_4)$

Learning an RNN-LM

- Each training example is a sequence (e.g. sentence), so we have training data D = $\{\underline{\mathbf{w}^{(1)}}, \mathbf{w}^{(2)}, \ldots, \mathbf{w}^{(N)}\}$ $= \{w_1^{(1)}, w_2^{(1)}, \ldots, w_T^{(1)}]$
- The objective function for a Deep LM (e.g. RNN-LM or Tranformer-LM) is typically the log-likelihood of the training examples:
  $$J(\boldsymbol{\theta}) = \Sigma_i \log p_{\boldsymbol{\theta}}(\mathbf{w}^{(i)})$$
- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots, w_T)$$
$$= \log p(w_1 \mid h_1) + \log p(w_2 \mid h_2) + \ldots + \log p(w_2 \mid h_T)$$

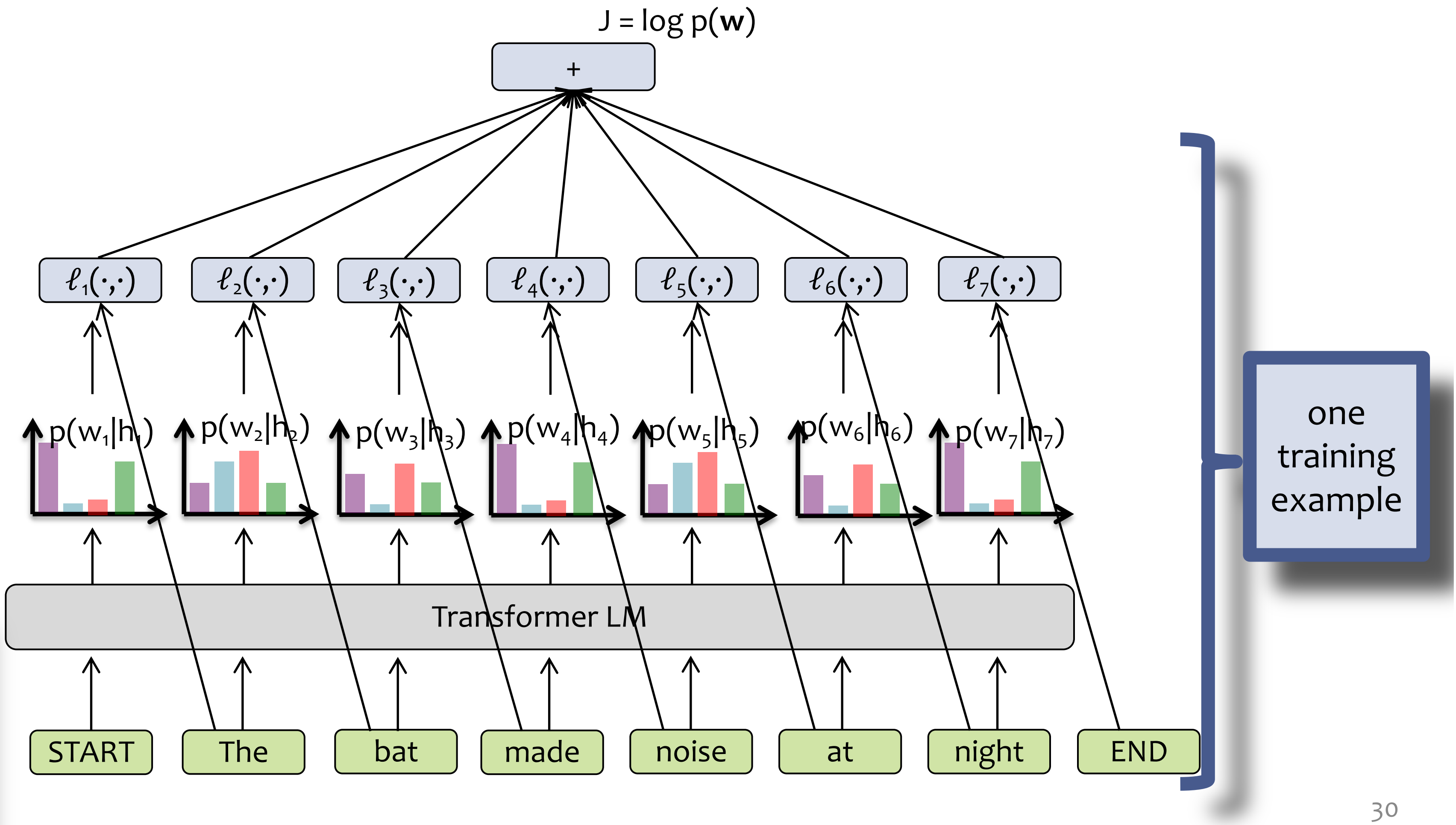


Learning a Transformer LM

- Each training example is a sequence (e.g. sentence), so we have training data D = {$\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$, … , $\mathbf{w}^{(N)}$}
- The objective function for a Deep LM (e.g. RNN-LM or Tranformer-LM) is typically the log-likelihood of the training examples:
  $$J(\boldsymbol{\theta}) = \Sigma_i \log p_{\boldsymbol{\theta}}(\mathbf{w}^{(i)})$$
- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

Training a Transformer-LM is the same, except we swap in a different deep language model.

$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots, w_T)$$
$$= \log p(w_1 \mid h_1) + \log p(w_2 \mid h_2) + \ldots + \log p(w_2 \mid h_T)$$

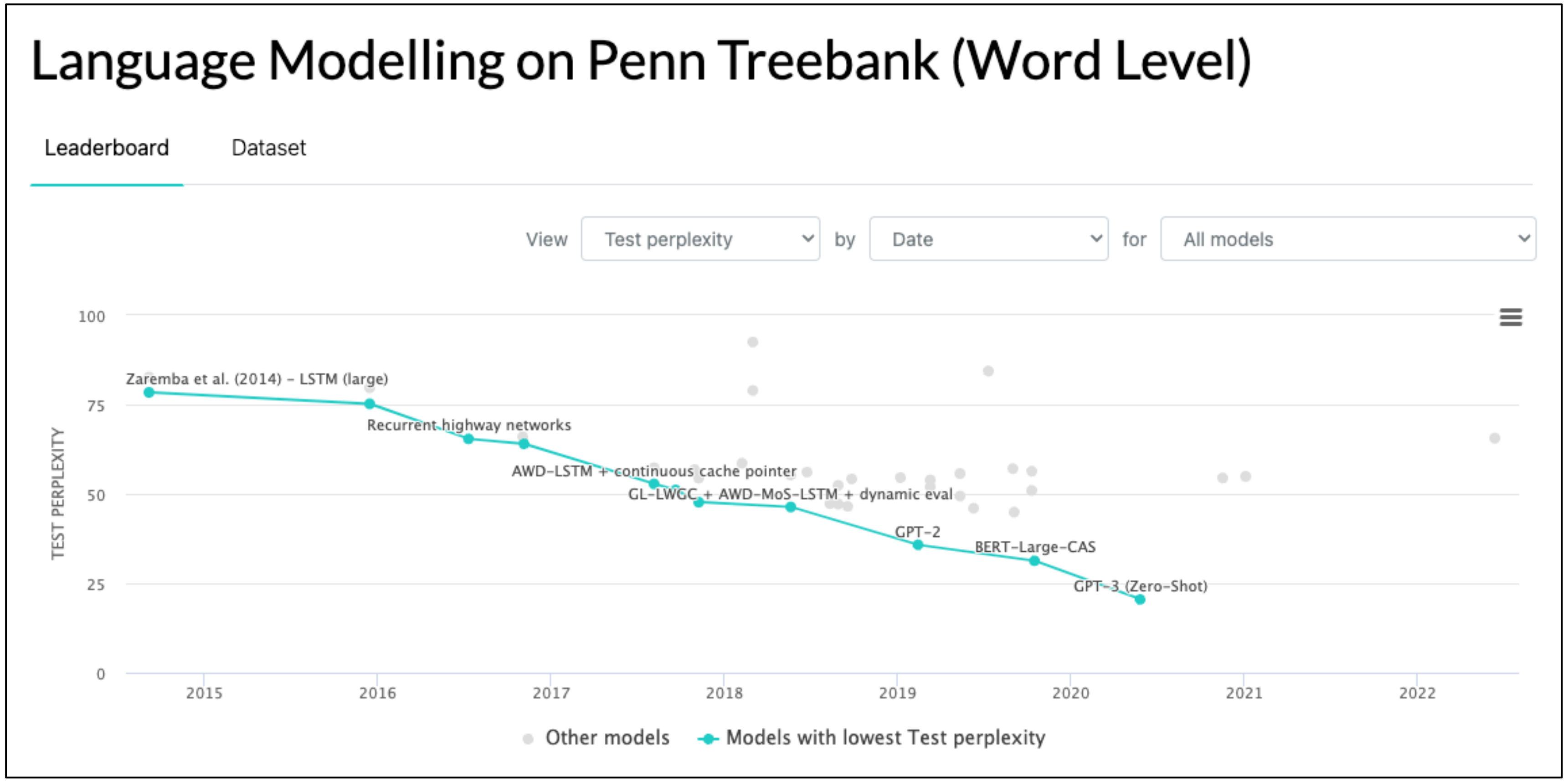


Language Modeling

An aside:

- State-of-the-art language models currently tend to rely on **transformer networks** (e.g. GPT-2)
- RNN-LMs comprised most of the early neural LMs that **led to** current SOTA architectures

$2^{-\log p(\vec{w})}$


Language Modelling on Penn Treebank (Word Level)

Leaderboard | Dataset

View [Test perplexity] by [Date] for [All models]

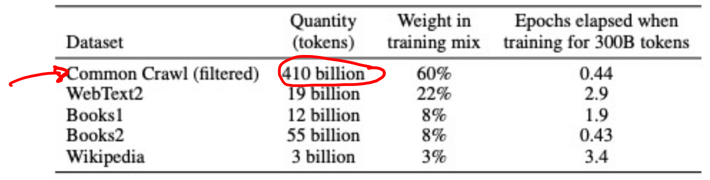
Figure from https://paperswithcode.com/sota/language-modelling-on-penn-treebank-word

EFFICIENT TRANSFORMERS

Why does efficiency matter?

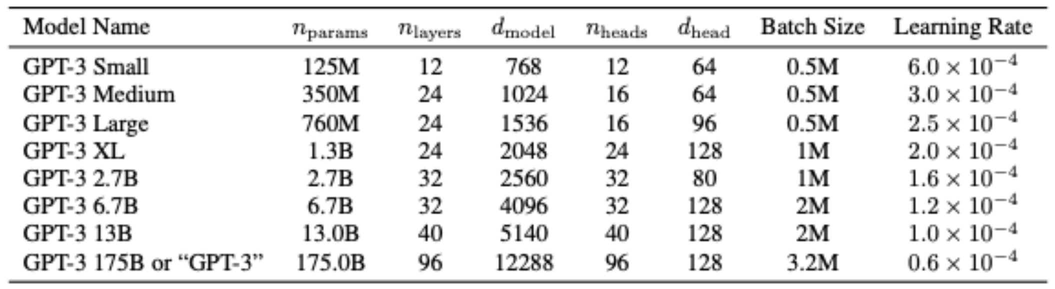
Case Study: GPT-3

- # of training tokens = 500 billion
- # of parameters = 175 billion
- # of cycles = 50 petaflop/s-days (each of which are 8.64e+19 flops)



Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.



Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

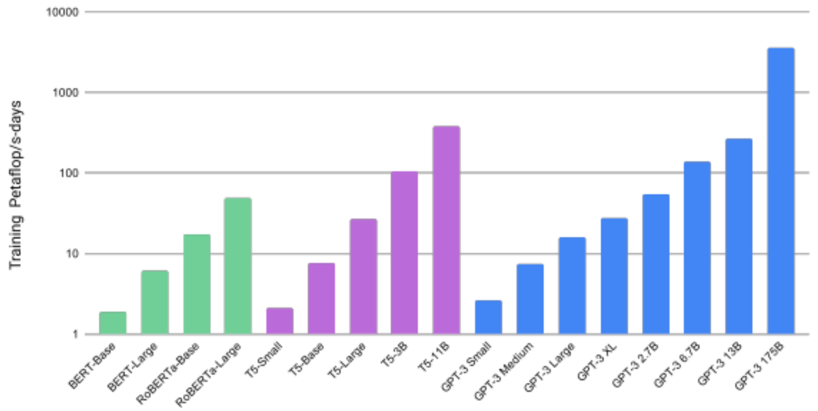


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

Figure from https://arxiv.org/pdf/2005.14165

Efficient Parallelism for Transformers

Transformers can be trained very efficiently!

(This is arguably one of the key reasons they have been so successful.)

- **Batching**: Rather than processing one sentence at a time, Transformers take in a batch of B sentences at a time. The computation is identical for each batch and is trivially parallelized.
- **Scaled Dot-product Attention**: can be easily parallelized because the attention scores of one timestep do not depend on other timesteps.
- **Multi-headed Attention**: computes each head independently, which permits yet more parallelism.
- **Matrix multiplication:** The core computation in attention is matrix multiplication, and specialized hardware (GPUs and TPUs) makes this very fast.
- **Model parallelism:** For huge models, we can divide the model over multiple GPUs/machines.
- **Key-value caching:** The keys and values are re-used over many timesteps, but we do not need to cache the queries, similarity scores, and attention weights.

Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We set our block size (maximum sequence length) to 10
- Before collecting them into a batch, we:
  1. truncate those sentences that are too long
  2. pad the sentences that are too short
  3. convert each token to an integer via a lookup table (vocabulary)
  4. convert each token to an embedding vector of fixed length

i	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}
1	In	the	hole	in	the	ground	there	lived	a	hobbit		
2	It	is	our	choices	that	show	what	we	truly	are		
3	It	was	the	best	of	times	it	was	the	worst	of	times
4	Even	miracles	take	a	little	time						
5	The	more	that	you	read	the	more	things	you	will	know	
6	We'll	always	have	each	other	no	matter	what	happens			
7	The	sun	did	not	shine	it	was	too	wet	to	play	
8	The	important	thing	is	to	never	stop	questioning				

Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We set our block size (maximum sequence length) to 10
- Before collecting them into a batch, we:
  1. truncate those sentences that are too long
  2. pad the sentences that are too short
  3. convert each token to an integer via a lookup table (vocabulary)
  4. convert each token to an embedding vector of fixed length

i	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}
1	In	the	hole	in	the	ground	there	lived	a	hobbit		
2	It	is	our	choices	that	show	what	we	truly	are		
3	It	was	the	best	of	times	it	was	the	worst	of	times
4	Even	miracles	take	a	little	time	<PAD>	<PAD>	<PAD>	<PAD>		
5	The	more	that	you	read	the	more	things	you	will	know	
6	We'll	always	have	each	other	no	matter	what	happens	<PAD>		
7	The	sun	did	not	shine	it	was	too	wet	to	play	
8	The	important	thing	is	to	never	stop	questioning	<PAD>	<PAD>		

Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We set our block size (maximum sequence length) to 10
- Before collecting them into a batch, we:
  1. truncate those sentences that are too long
  2. pad the sentences that are too short
  3. convert each token to an integer via a lookup table (vocabulary)
  4. convert each token to an embedding vector of fixed length

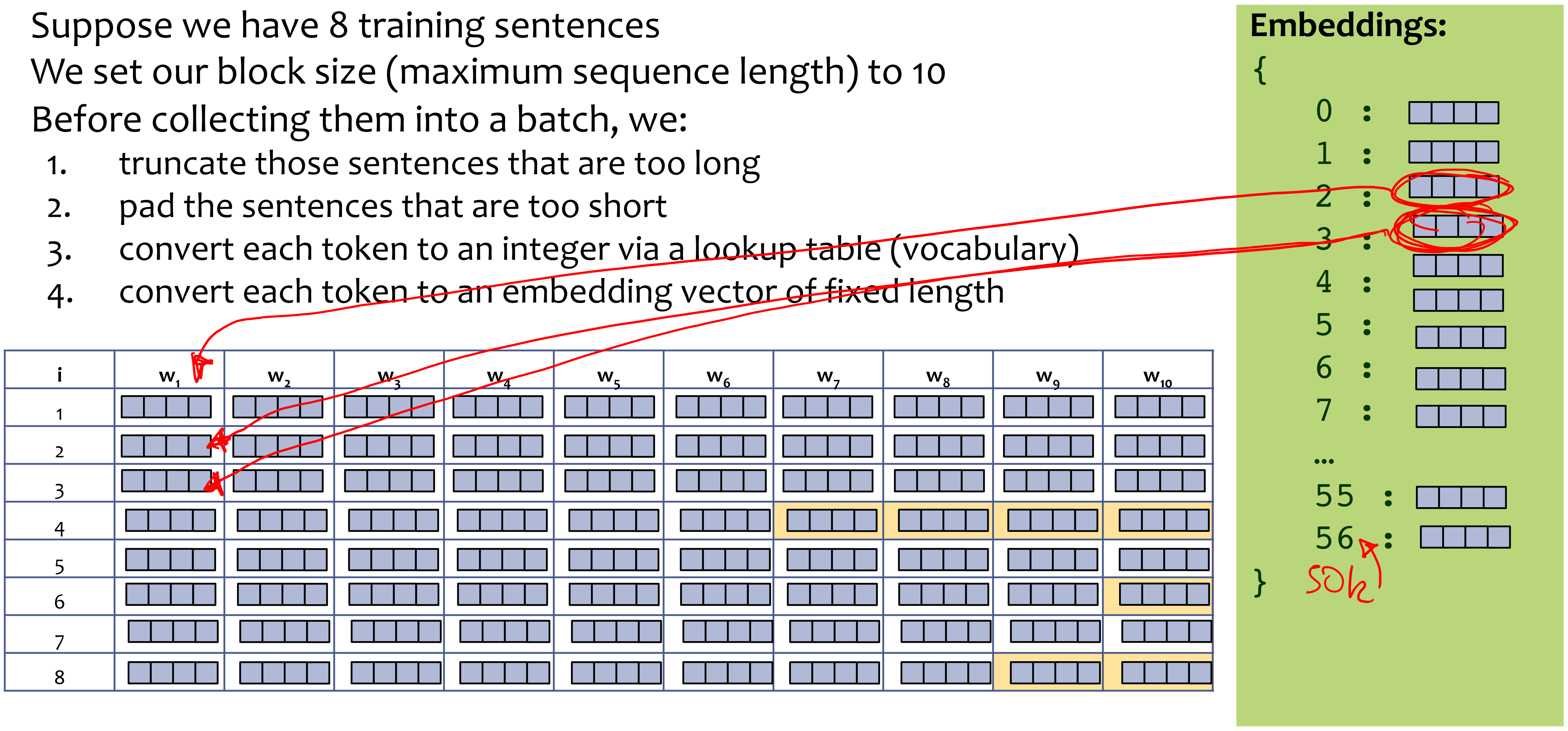
i	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}
1	2	41	17	19	41	13	42	23	6	16
2	3	20	32	10	40	36	53	51	49	8
3	3	50	41	9	30	46	21	50	41	55
4	1	25	39	6	22	45	0	0	0	0
5	4	26	40	56	34	41	26	44	56	54
6	5	7	15	12	31	28	24	53	14	0
7	4	38	11	29	35	21	50	48	52	47
8	4	18	43	20	47	27	37	33	0	0

Vocabulary:

```
{
    '<PAD>': 0,
    'Even': 1,
    'In': 2,
    'It': 3,
    'The': 4,
    "We'll": 5,
    'a': 6,
    'always': 7,
    'are': 8,
    'best': 9,
    …
    'what': 53,
    'will': 54,
    'worst': 55,
    'you': 56
}
```

Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We set our block size (maximum sequence length) to 10
- Before collecting them into a batch, we:
  1. truncate those sentences that are too long
  2. pad the sentences that are too short
  3. convert each token to an integer via a lookup table (vocabulary)
  4. convert each token to an embedding vector of fixed length



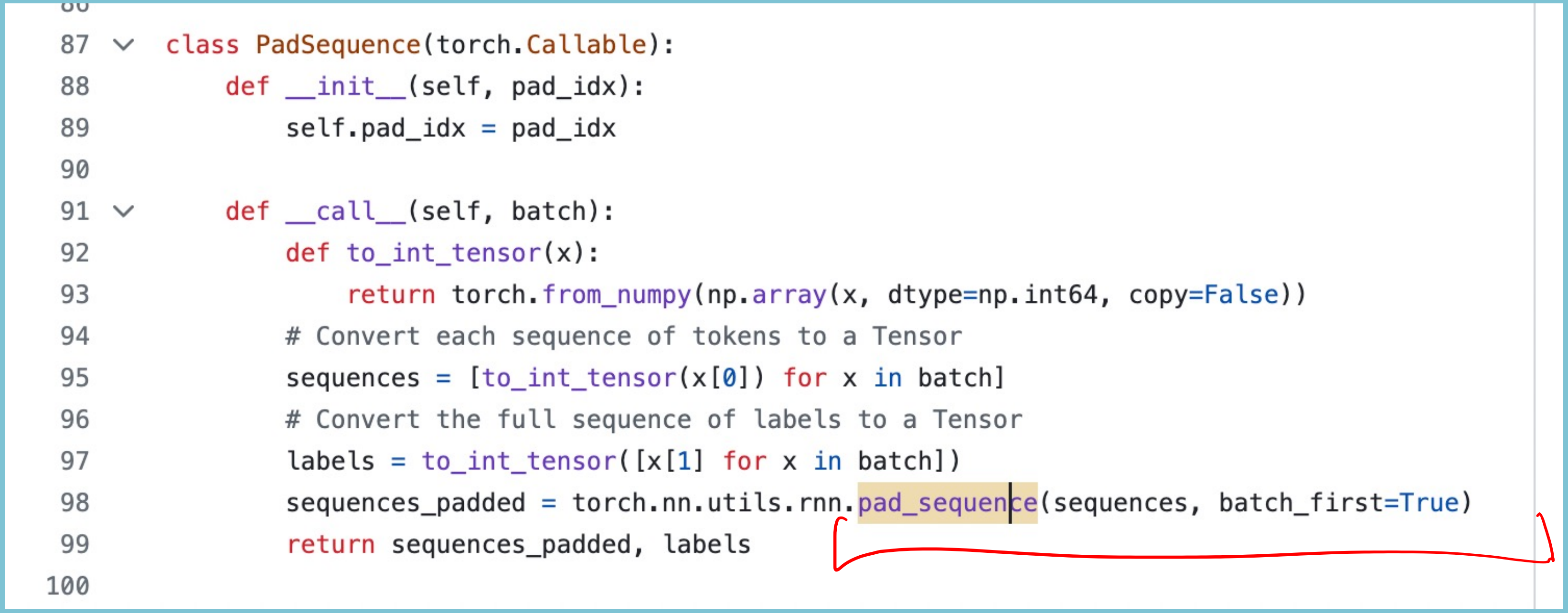
Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We s
- Befo
  1.
  2.
  3.
  4.

i
1
2
3
4
5
6
7
8

Don’t do this!

(Can you spot the bug?)



```python
class PadSequence(torch.Callable):
    def __init__(self, pad_idx):
        self.pad_idx = pad_idx

    def __call__(self, batch):
        def to_int_tensor(x):
            return torch.from_numpy(np.array(x, dtype=np.int64, copy=False))
        # Convert each sequence of tokens to a Tensor
        sequences = [to_int_tensor(x[0]) for x in batch]
        # Convert the full sequence of labels to a Tensor
        labels = to_int_tensor([x[1] for x in batch])
        sequences_padded = torch.nn.utils.rnn.pad_sequence(sequences, batch_first=True)
        return sequences_padded, labels
```

Efficient Parallelism for Transformers

Transformers can be trained very efficiently!

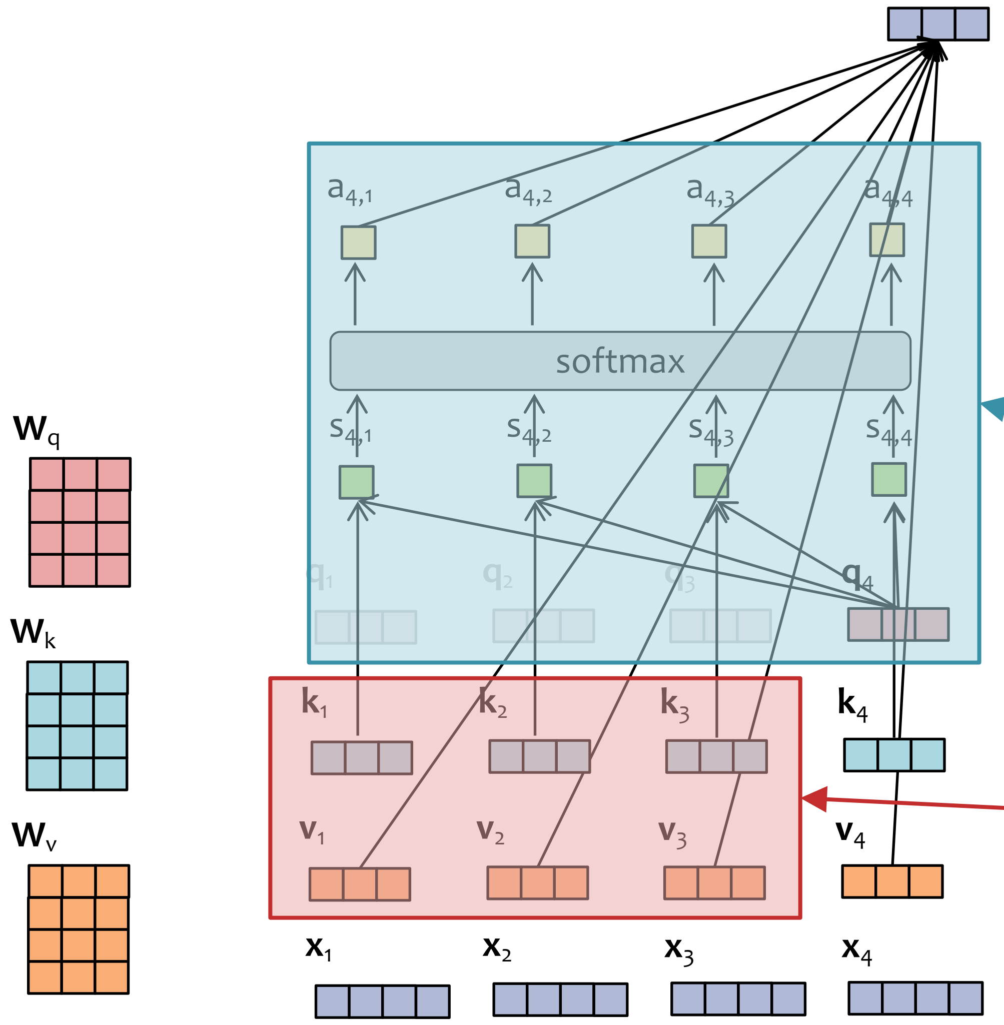
(This is arguably one of the key reasons they have been so successful.)

- **Batching**: Rather than processing one sentence at a time, Transformers take in a batch of B sentences at a time. The computation is identical for each batch and is trivially parallelized.
- **Scaled Dot-product Attention**: can be easily parallelized because the attention scores of one timestep do not depend on other timesteps.
- **Multi-headed Attention**: computes each head independently, which permits yet more parallelism.

$$X' = \text{softmax}\left(\underbrace{QK^T}/\sqrt{d_k}\right)V$$

- **Matrix multiplication:** The core computation in attention is matrix multiplication, and specialized hardware (GPUs and TPUs) makes this very fast.
- **Model parallelism:** For huge models, we can divide the model over multiple GPUs/machines.
- **Key-value caching**: The keys and values are re-used over many timesteps, but we do not need to cache the queries, similarity scores, and attention weights.

Key-Value Cache



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$$

$$s_{4,j} = \mathbf{k}_j^T\mathbf{q}_4/\sqrt{d_k}$$

$$\mathbf{q}_j = \mathbf{W}_q^T\mathbf{x}_j$$

$$\mathbf{k}_j = \mathbf{W}_k^T\mathbf{x}_j$$

$$\mathbf{v}_j = \mathbf{W}_v^T\mathbf{x}_j$$

- At each timestep, we reuse all previous keys and values (i.e. we need to cache them)
- But we can get rid of the queries, similarity scores, and attention weights (i.e. we can let them fall out of the cache)

Discarded after this timestep

Computed for previous time-steps and reused for this timestep

TOKENIZATION

Tokenization

Word-based Tokenizer:

Input: “Henry is giving a lecture on transformers”

Output: [“henry”, “is”, “giving”, “a”, “lecture”, “on”, “transformers”]

Pros/Cons:

- Can have difficulty trading off between vocabulary size and computational tractability
- Similar words e.g., “transformers” and “transformer” can get mapped to completely disparate representations
- Typos will typically be out-of-vocabulary (OOV)

Tokenization

Word-based Tokenizer:

Input: "Henry is givin' a lectrue on transformers"

Output: ["henry", "is", <OOV>, "a", <OOV>, "on", "transformers"]

Pros/Cons:

- Can have difficulty trading off between vocabulary size and computational tractability
- Similar words e.g., "transformers" and "transformer" can get mapped to completely disparate representations
- Typos will typically be out-of-vocabulary (OOV)

Tokenization

Character-based Tokenizer:

Input: “Henry is givin’ a lectrue on transformers”

Output: [“h”, “e”, “n”, “r”, “y”, “i”, “s”, “g”, “i”, “v”, “i”, “n”, “ ’ ”, …]

Pros/Cons:

- Much smaller vocabularies but a lot of semantic meaning is lost…
- Sequences will be much longer than word-based tokenization, potentially causing computational issues
- Can do well on logographic languages e.g., Kanji 漢字

Tokenization

Subword-based Tokenizer:

Input: “Henry is givin’ a lectrue on transformers”

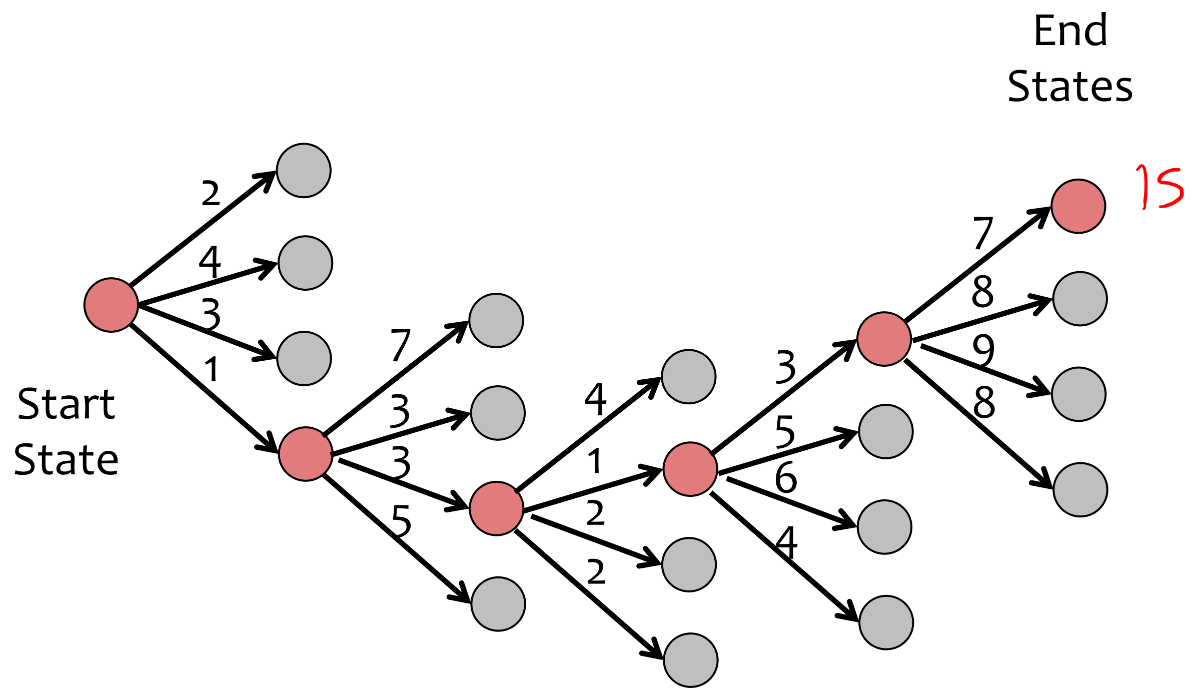
Output: [“henry”, “is”, “giv”, “##in”, “ ‘ ”, “a”, “lec” “##true”, “on”, “transform”, “##ers”]

Pros/Cons:

- Split long or rare words into smaller, semantically meaningful components or subwords
- No out-of-vocabulary words – any non-subword token can be constructed from other subwords (always includ all characters as subwords)
- Examples algorithms for learning a subword tokenization:
  - Byte-Pair-Encoding (BPE), WordPiece, SentencePiece

GREEDY DECODING FOR A LANGUAGE MODEL

Background: Greedy Search



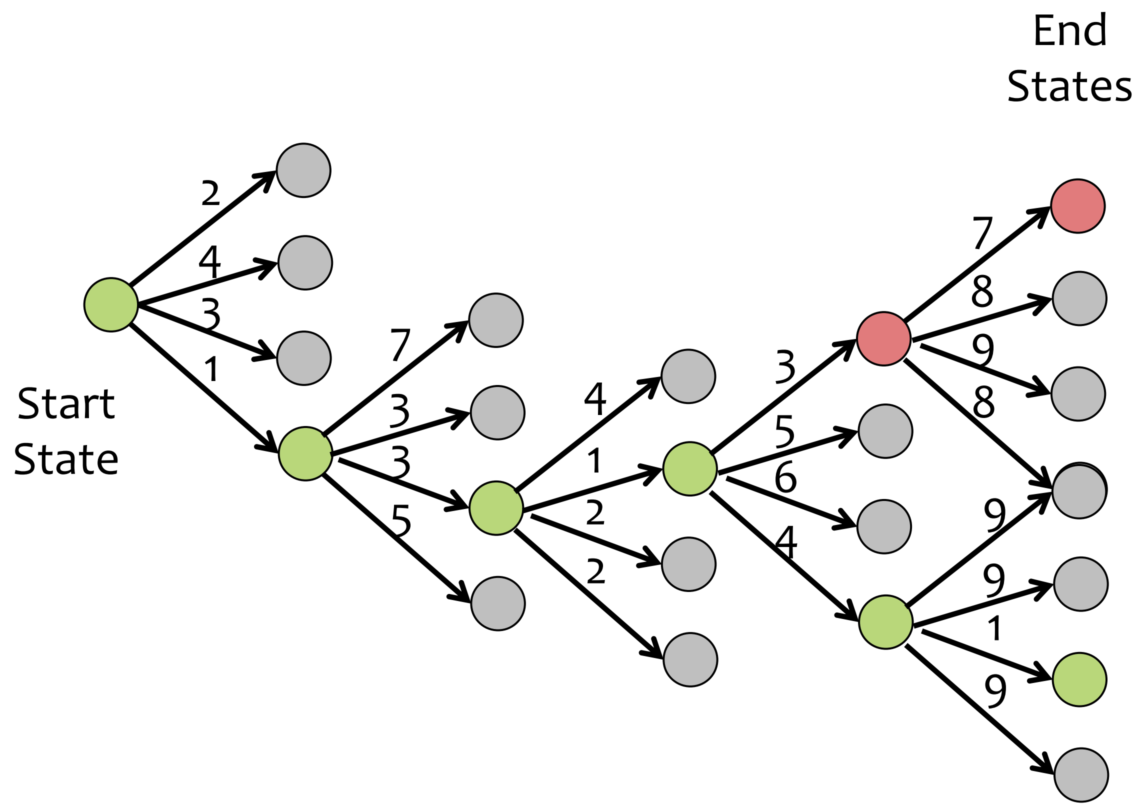
Goal:

- Search space consists of nodes and weighted edges
- Goal is to find the lowest (total) weight path from root to a leaf

Greedy Search:

- At each node, selects the edge with lowest (immediate) weight
- **Heuristic** method of search (i.e. does *not* necessarily find the best path)
- Computation time: **linear** in max path length

Background: Greedy Search



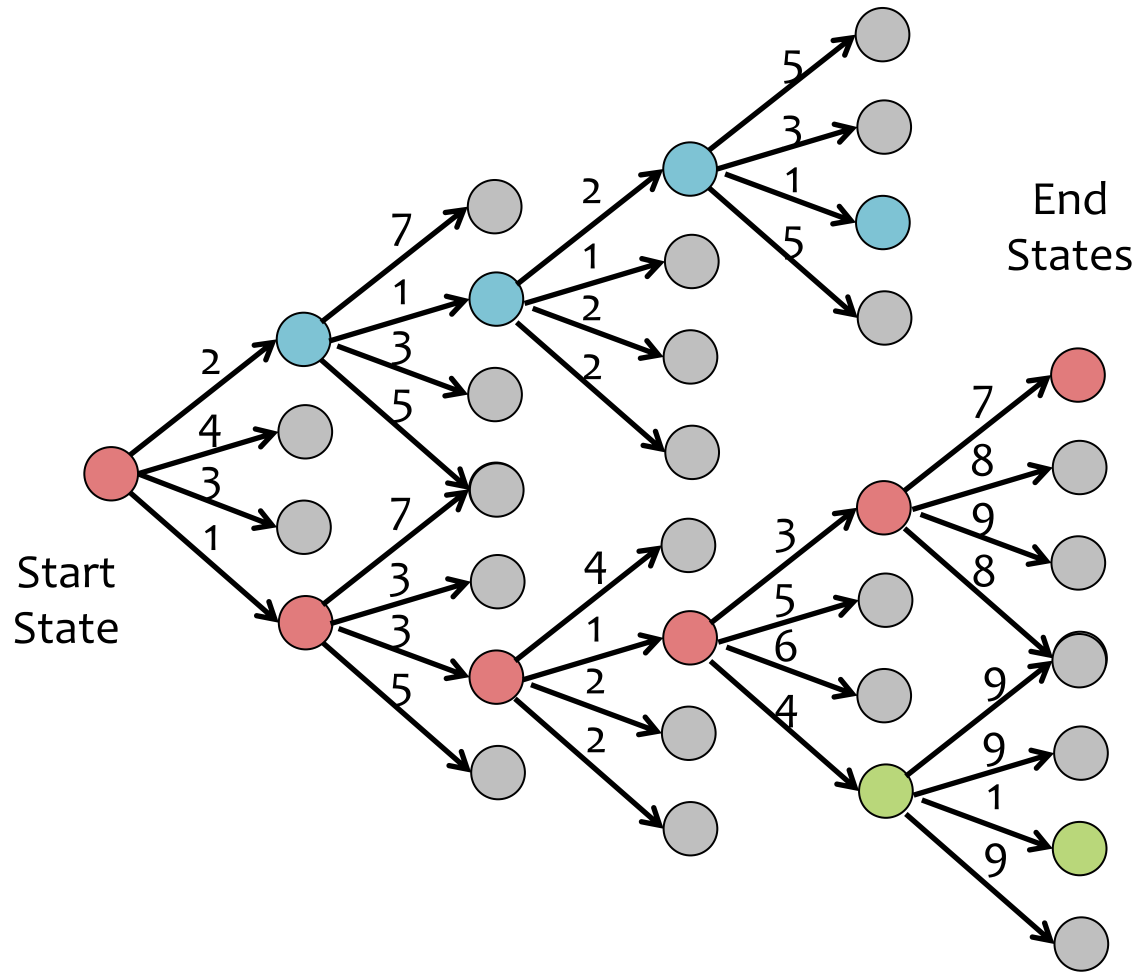
Goal:

- Search space consists of nodes and weighted edges
- Goal is to find the lowest (total) weight path from root to a leaf

Greedy Search:

- At each node, selects the edge with lowest (immediate) weight
- **Heuristic** method of search (i.e. does *not* necessarily find the best path)
- Computation time: **linear** in max path length

Background: Greedy Search



Goal:

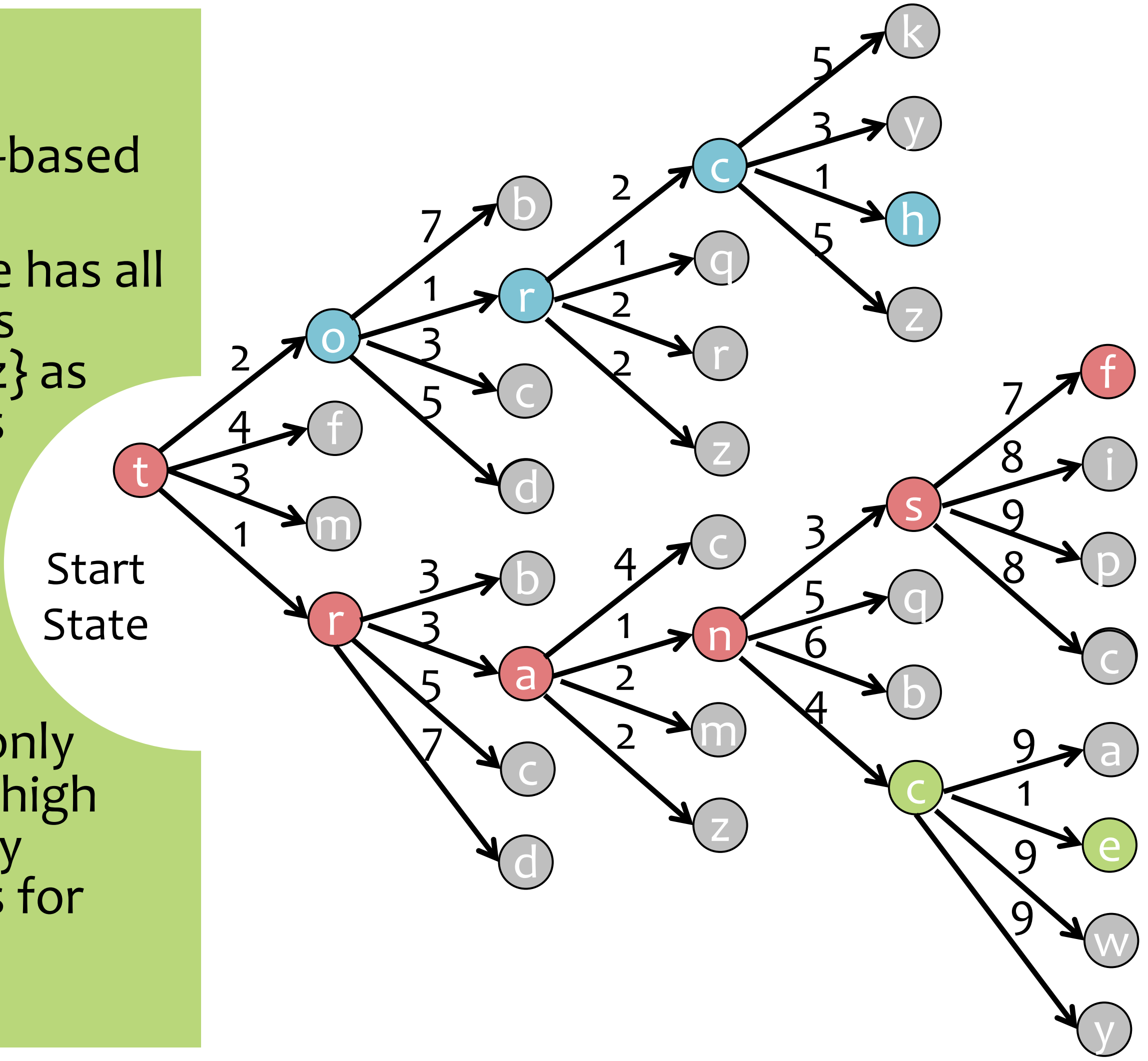
- Search space consists of nodes and weighted edges
- Goal is to find the lowest (total) weight path from root to a leaf

Greedy Search:

- At each node, selects the edge with lowest (immediate) weight
- **Heuristic** method of search (i.e. does *not* necessarily find the best path)
- Computation time: **linear** in max path length

Greedy Decoding for a Language Model

Setup:
- Assume a character-based tokenizer
- Each node has all characters {a,b,c,…,z} as neighbors
- Here we only show the high probability neighbors for space



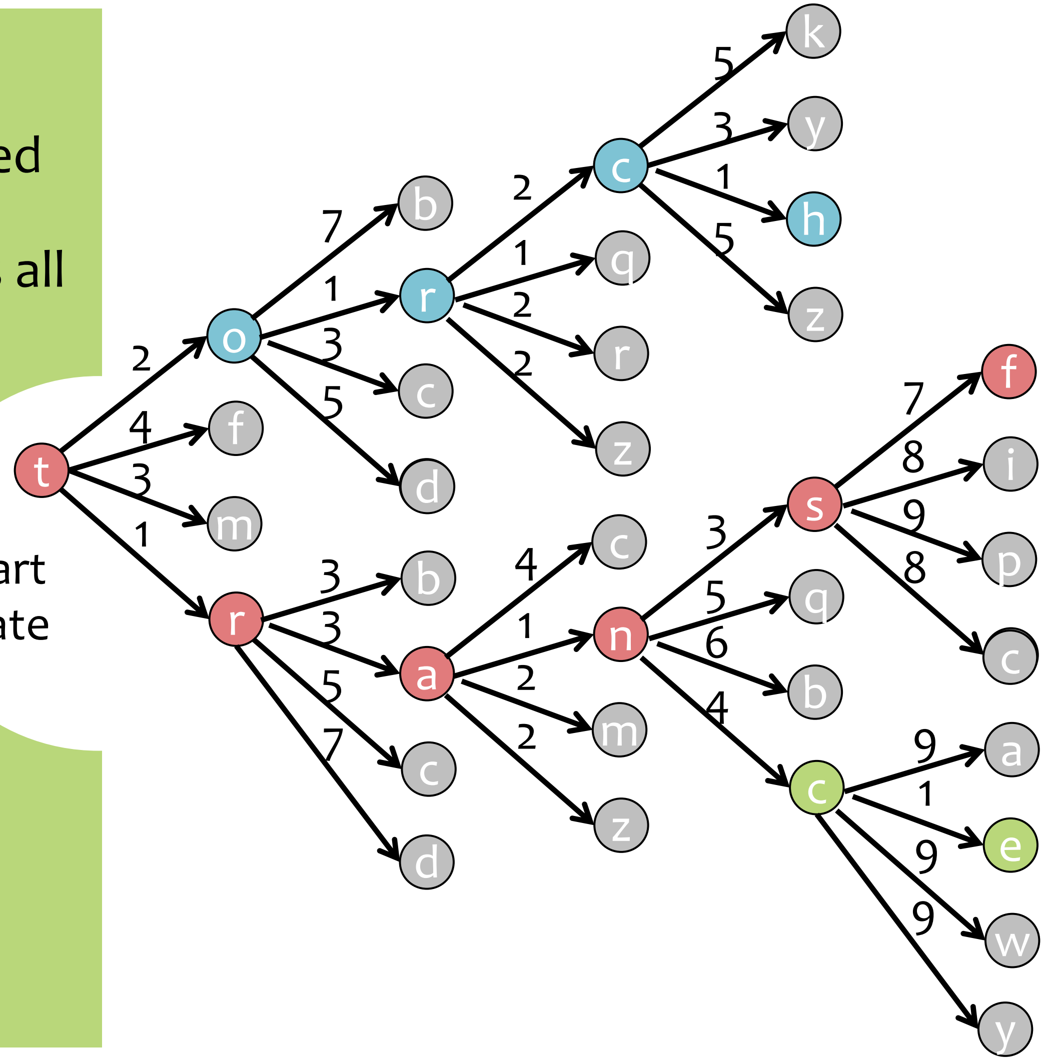
Goal:
- Search space consists of nodes (partial sentences) and weighted by negative log probability
- Goal is to find the highest probably (lowest negative log probability) path from root to a leaf

Greedy Search:
- At each node, selects the edge with lowest negative log probability
- **Heuristic** method of search (i.e. does *not* necessarily find the best path)
- Computation time: **linear** in max path length

Sampling from a Language Model

Setup:

- Assume a character-based tokenizer
- Each node has all characters {a,b,c,…,z} as neighbors
- Here we only show the high probability neighbors for space



Goal:

- Search space consists of nodes (partial sentences) and weighted by negative log probability
- Goal is to sample a path from root to a leaf with probability according to the probability of that path

Ancestral Sampling:

- At each node, randomly pick an edge with probability (converting from negative log probability)
- **Exact** method of sampling, assuming a locally normalized distribution (i.e. does *not* necessarily find the best path)
- Computation time: **linear** in max path length