



# 10-423/10-623 Generative AI

Machine Learning Department  
School of Computer Science  
Carnegie Mellon University

# Learning Large Language Models and Decoding

Pat Virtue and Matt Gormley

Lecture 3

Jan. 22, 2025

# A Recipe for Machine Learning

Recall...

1. Given training data:

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

2. Choose each of these:

– Decision function

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

– Loss function

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

3. Define goal:

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

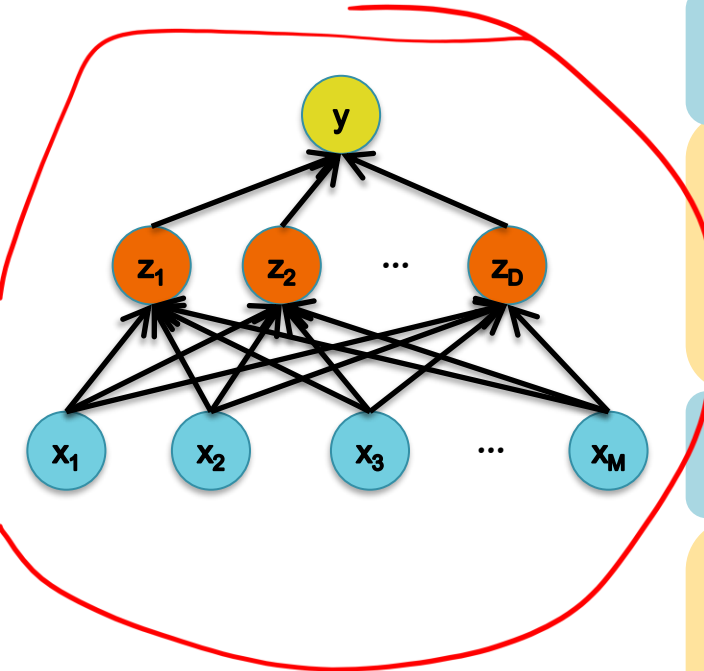
4. Train with SGD:

(take small steps opposite the gradient)

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

# Backpropagation

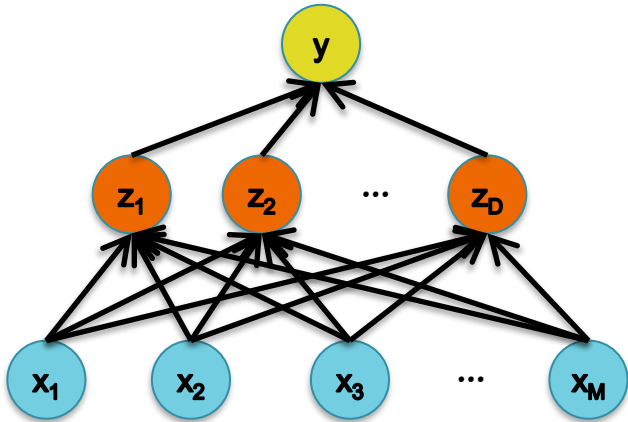
Example:  
Neural Network



	Forward	Backward
Loss	$J = y^* \log y + (1 - y^*) \log(1 - y)$	$g_y = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
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

Example:  
Neural Network



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

Backward

Loss

$$g_y = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

Sigmoid

$$g_b = g_y \frac{\partial y}{\partial b}, \quad \frac{\partial y}{\partial b} = y(1 - y)$$

Linear

$$g_{\beta_j} = g_b \frac{\partial b}{\partial \beta_j}, \quad \frac{\partial b}{\partial \beta_j} = z_j$$

Sigmoid

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$g_{z_j} = g_b \frac{\partial b}{\partial z_j}, \quad \frac{\partial b}{\partial z_j} = \beta_j$$

$$g_{a_j} = g_{z_j} \frac{\partial z_j}{\partial a_j}, \quad \frac{\partial z_j}{\partial a_j} = z_j(1 - z_j)$$

Linear

$$a_j = \sum_{i=0}^M \alpha_{ji} x_i$$

$$g_{\alpha_{ji}} = g_{a_j} \frac{\partial a_j}{\partial \alpha_{ji}}, \quad \frac{\partial a_j}{\partial \alpha_{ji}} = x_i$$

$$g_{x_i} = \sum_{j=0}^D g_{a_j} \frac{\partial a_j}{\partial x_i}, \quad \frac{\partial a_j}{\partial x_i} = \alpha_{ji}$$

# LEARNING A TRANSFORMER LM

# Language Models

Vocab: 35 unique tokens  
950 total tokens

1. Given training data:

$$\{\mathbf{x}_i \rightarrow \mathbf{y}_i\}_{i=1}^N$$

2. Choose Decision function

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

## Data

$i=1$   
 $i=2$

I am Sam .

I am Sam .

Sam I am .

That Sam-I-am .

That Sam-I-am !

I do not like that Sam-I-am

Do you like green eggs and ham

I do not like them , Sam-I-am .

I do not like green eggs and ham .

Would you like them here or there ?

I would not like them here or there .

I would not like them anywhere .

I do not like green eggs and ham .

I do not like them , Sam-I-am .

Would you like them in a house ?

Would you like them with a mouse ?

I do not like them in a house .

I do not like them with a mouse .

I do not like them here or there

But what is the task?

$x_1, x_2, x_3 \rightarrow x_4$

$i=N$

# Language Models

Vocab: 35 unique tokens  
950 total tokens

1. Given training data:

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

Input → Output

START →  $x_1^{(1)}$

START,  $x_1^{(1)}$  →  $x_2^{(1)}$

START,  $x_{1:T-1}$  →  $x_T^{(1)}$

$T$  training pairs for each sequence  $x^{(i)}$

$i=1$   
I am Sam .  
 $x_1^{(1)}$   $x_2^{(1)}$   $x_3^{(1)}$   $x_4^{(1)}$

I

I am

I am Sam

I am Sam .

## Data

I am Sam .

I am Sam .

Sam I am .

That Sam-I-am .

That Sam-I-am !

I do not like that Sam-I-am

Do you like green eggs and ham

I do not like them , Sam-I-am .

I do not like green eggs and ham .

Would you like them here or there ?

I would not like them here or there .

I would not like them anywhere .

I do not like green eggs and ham .

I do not like them , Sam-I-am .

Would you like them in a house ?

Would you like them with a mouse ?

I do not like them in a house .

I do not like them with a mouse .

I do not like them here or there .

# Language Models

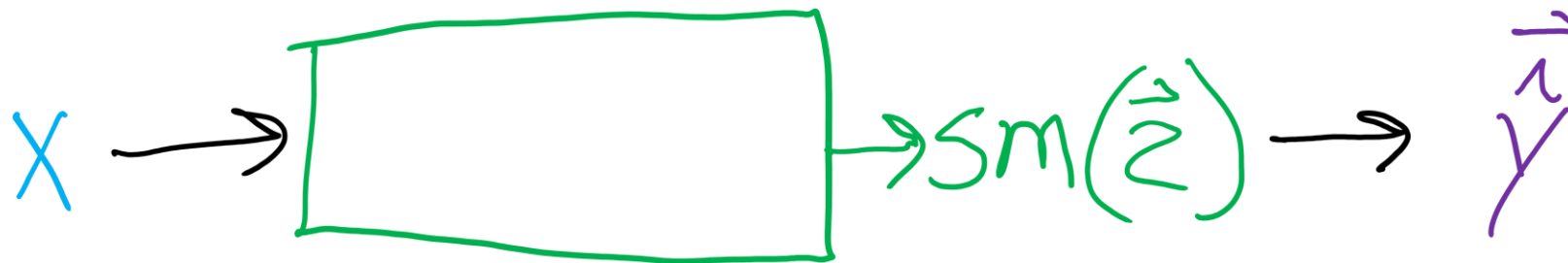
2. Choose Decision function

$$\hat{y} = f_{\theta}(x_i)$$

x  
"I do not"



$$\vec{y} \in [0, 1]^{|V|}$$



size of vocab  
|V|



# Language Models

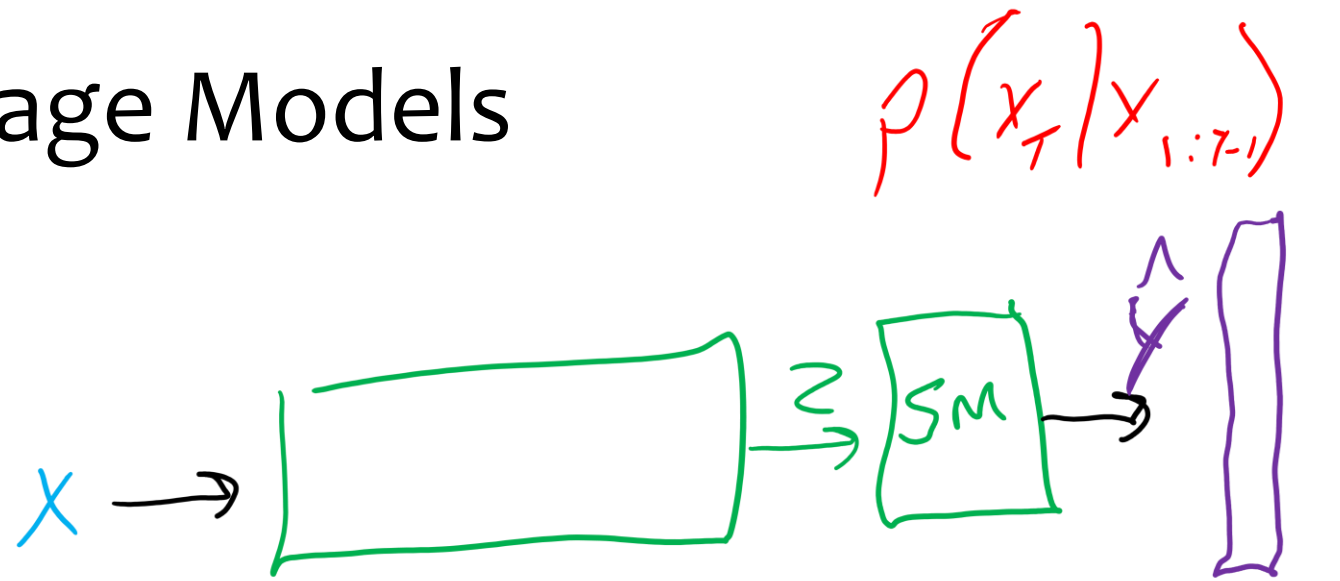
2. Choose each of these:

- Decision function

$$\hat{y} = f_{\theta}(x_i)$$

- Loss function

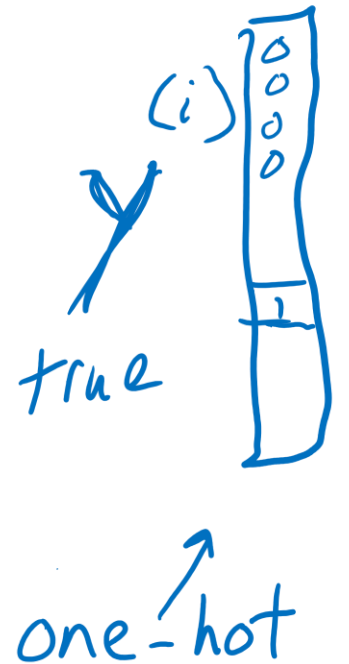
$$l(\hat{y}, y_i) \in \mathbb{R}$$



cross entropy



$$J(\theta) = \sum_{i=1}^N \sum_{t=1}^T \left( \sum_{k=1}^{|V|} y_k \log \hat{y}_k \right)$$



# Language Models

1. Given training data:

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

2. Choose each of these:

– Decision function

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

– Loss function

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

3. Define goal:

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

4. Train with SGD:

(take small steps  
opposite the gradient)

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

# **EFFICIENT TRANSFORMERS**

# Why does efficiency matter?

## Case Study: GPT-3

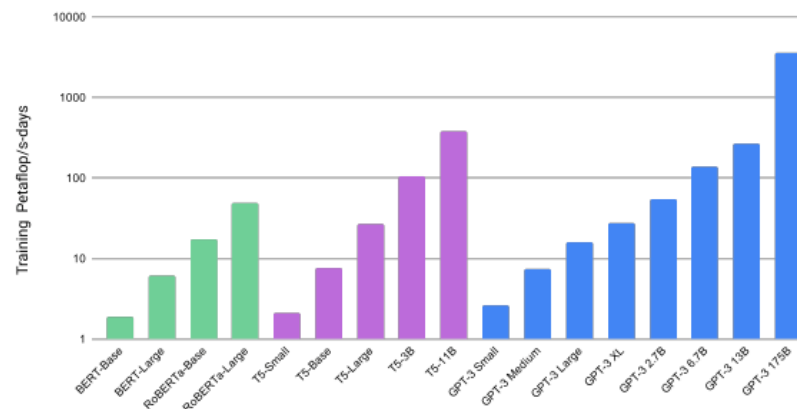
- # of training tokens = 500 billion
- # of parameters = 175 billion
- # of cycles = 50 petaflop/s-days (each of which are  $8.64e+19$  flops)

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

**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<sup>+</sup>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.

# 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 <sub>1</sub>	w <sub>2</sub>	w <sub>3</sub>	w <sub>4</sub>	w <sub>5</sub>	w <sub>6</sub>	w <sub>7</sub>	w <sub>8</sub>	w <sub>9</sub>	w <sub>10</sub>	w <sub>11</sub>	w <sub>12</sub>
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 <sub>1</sub>	w <sub>2</sub>	w <sub>3</sub>	w <sub>4</sub>	w <sub>5</sub>	w <sub>6</sub>	w <sub>7</sub>	w <sub>8</sub>	w <sub>9</sub>	w <sub>10</sub>
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

i	w <sub>1</sub>	w <sub>2</sub>	w <sub>3</sub>	w <sub>4</sub>	w <sub>5</sub>	w <sub>6</sub>	w <sub>7</sub>	w <sub>8</sub>	w <sub>9</sub>	w <sub>10</sub>
1										
2										
3										
4										
5										
6										
7										
8										

**Embeddings:**

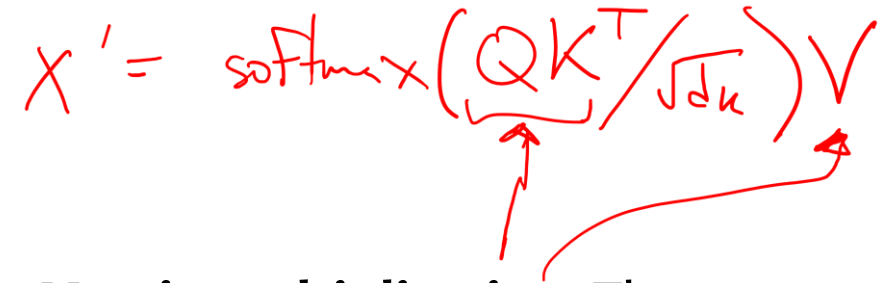
```
{
  0 : 
  1 : 
  2 : 
  3 : 
  4 : 
  5 : 
  6 : 
  7 : 
  ...
  55 : 
  56 : 
}
```

*SOH*

# Efficient Parallelism for Transformers

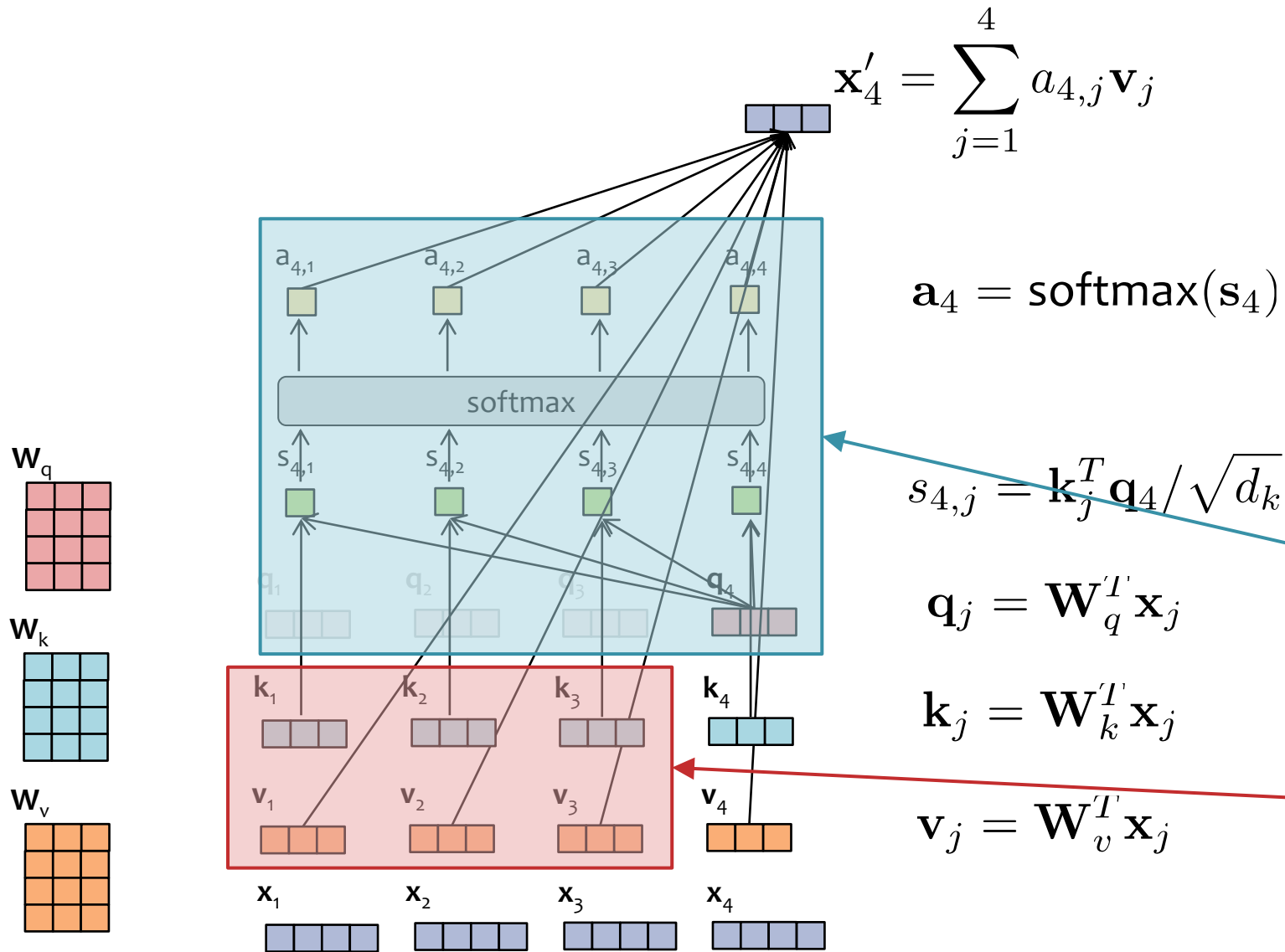
Transformers can be trained very efficiently!  
(This is arguably one of the key reasons they have been so successful.)

- **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.

$$X' = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$
The equation is written in red ink. A bracket underlines the term  $QK^T$ , and a red arrow points from the bottom of this bracket to the first bullet point. Another red arrow points from the  $V$  term to the second bullet point.

- **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



- 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”

The input text is shown on a light green background. Red arrows point from the words 'givin'', 'lectrue', and 'transformers' in the input to the corresponding tokens '<OOV>', '<OOV>', and 'transformers' in the output on the light blue background below.

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”]

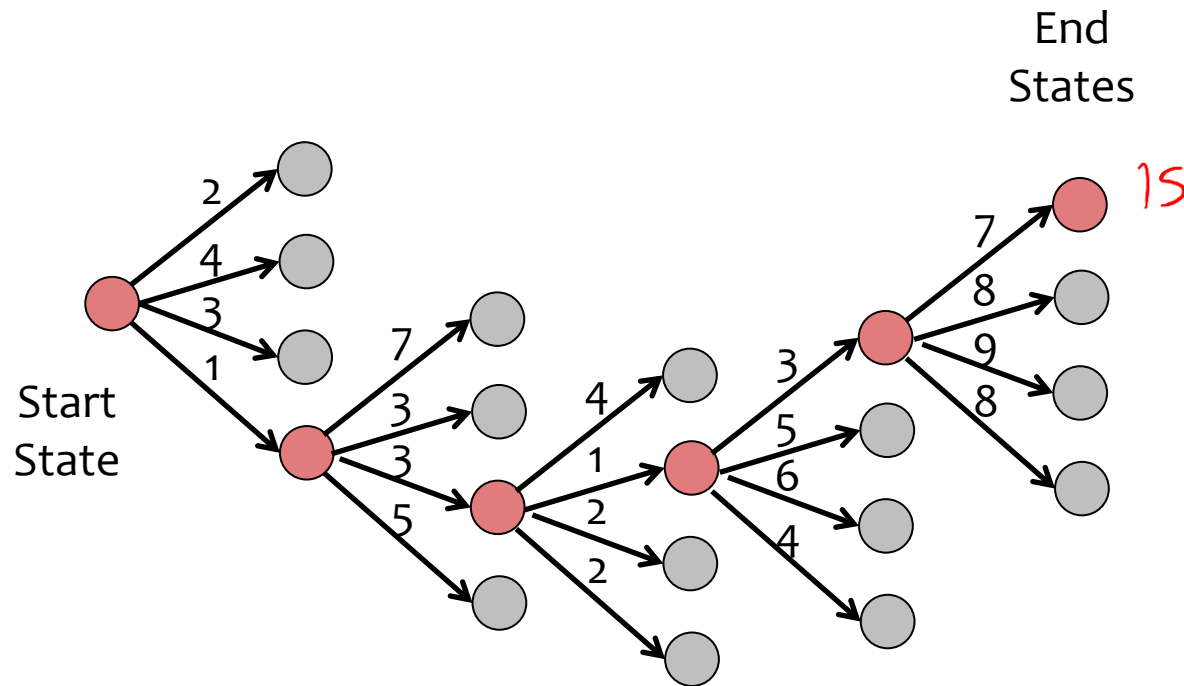
## Pros/Cons:

- 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



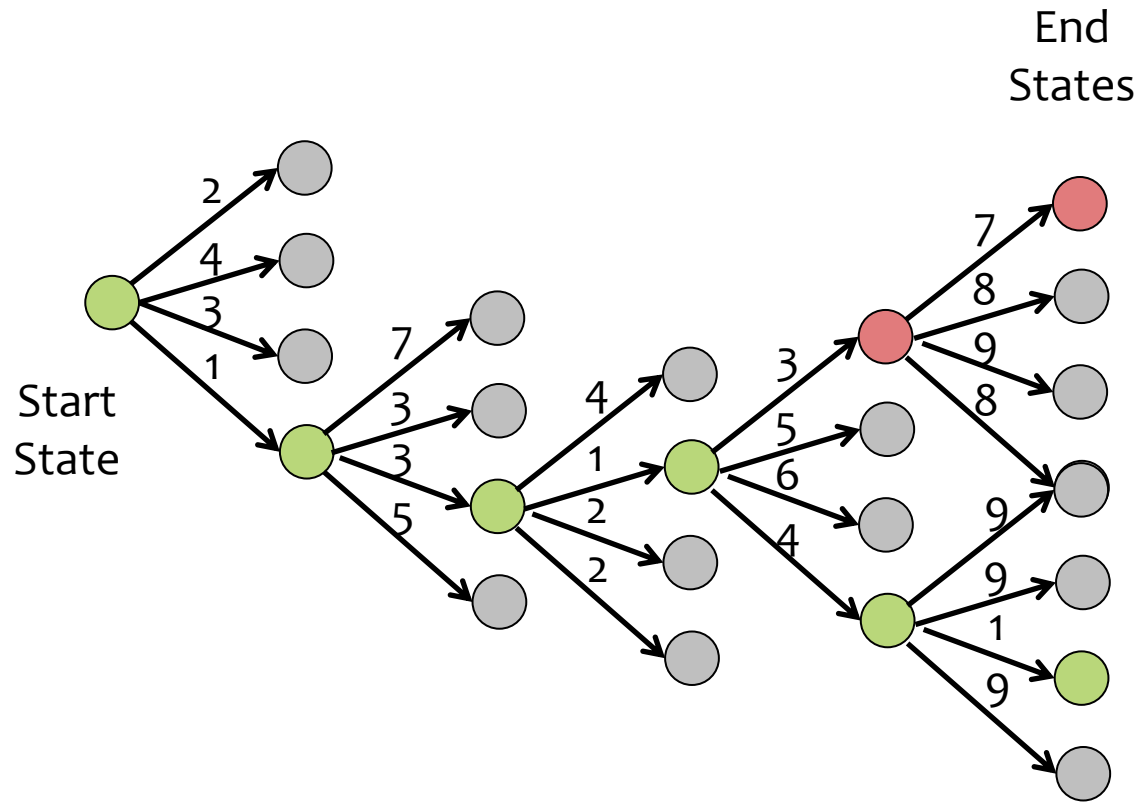
## Goal:

- 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



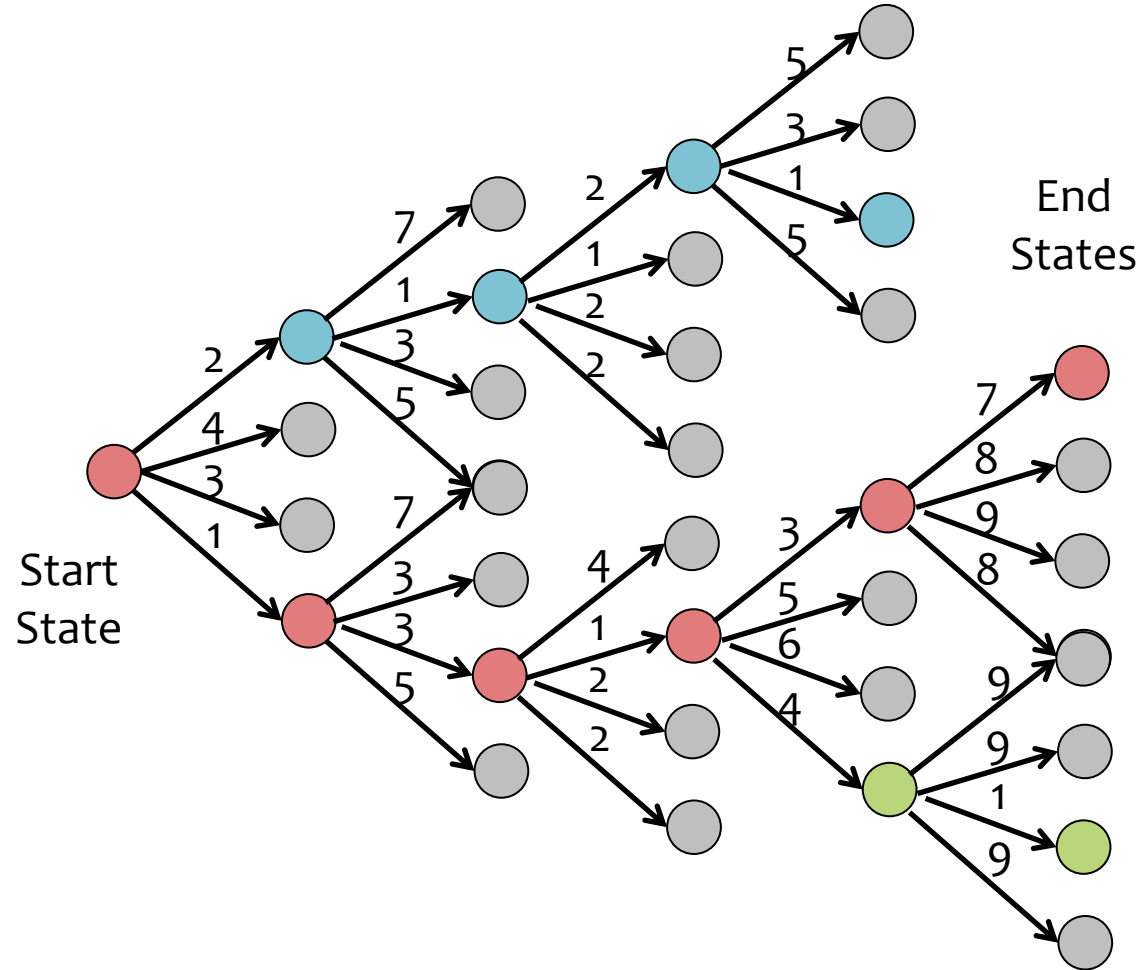
## Goal:

- 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:

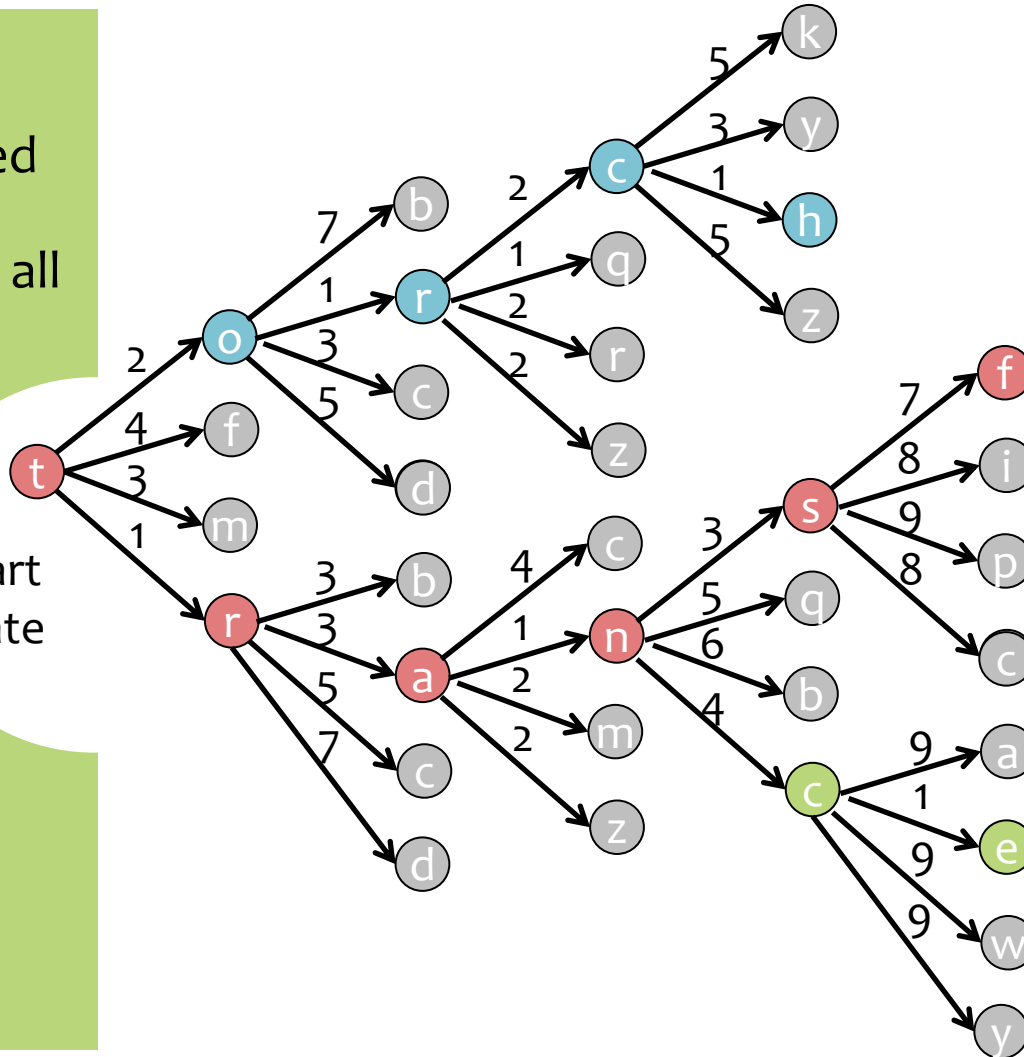
- At each node, selects the edge with lowest (immediate) weight
- **Heuristic** method of search (i.e. does not necessarily find the best path)
- Computation time: **linear** in max path length

# Greedy Decoding for a Language Model

## Setup:

- Assume a character-based tokenizer
- Each node has all characters {a,b,c,...,z} as neighbors

Start State



- 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:

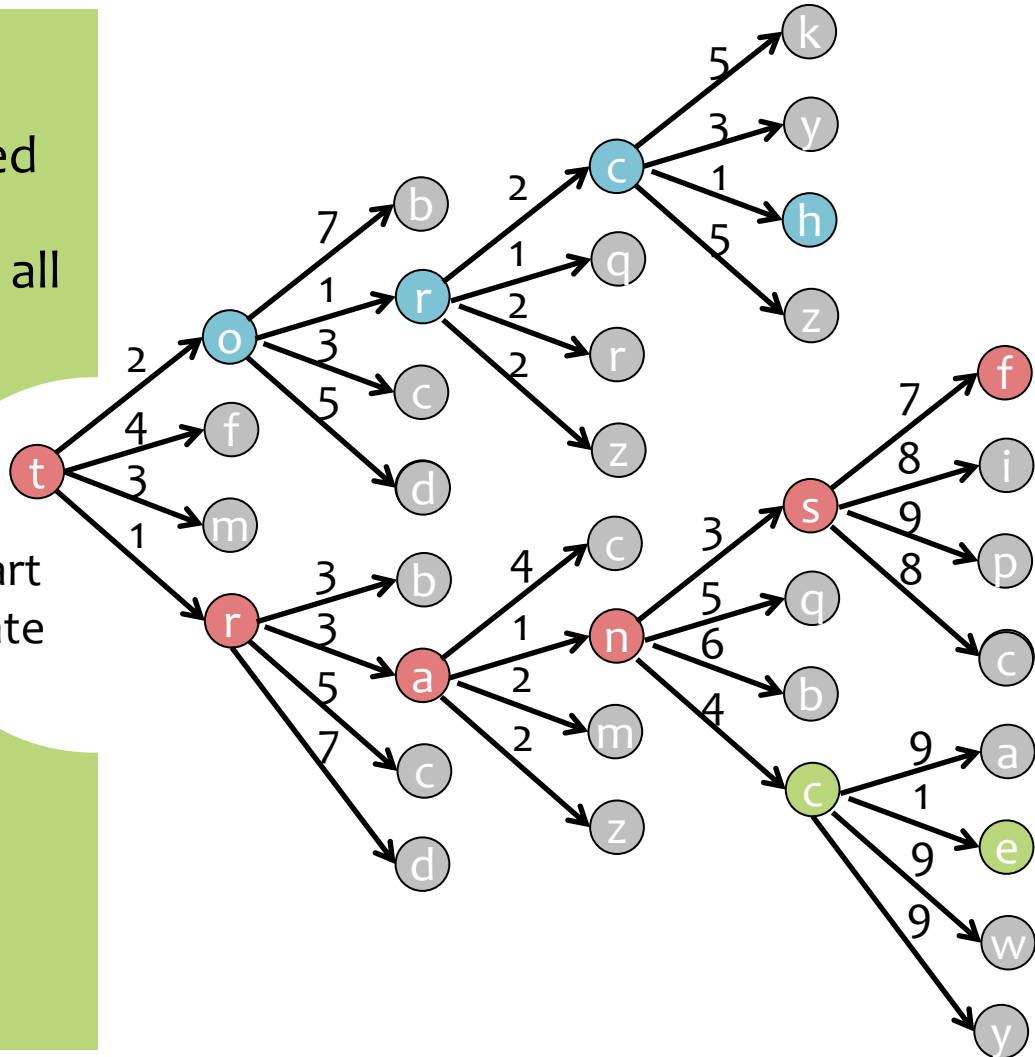
- At each node, selects the edge with lowest negative log probability
- **Heuristic** method of search (i.e. does not necessarily find the best path)
- Computation time: **linear** in max path length

# Sampling from a Language Model

## Setup:

- Assume a character-based tokenizer
- Each node has all characters {a,b,c,...,z} as neighbors

Start State



- 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