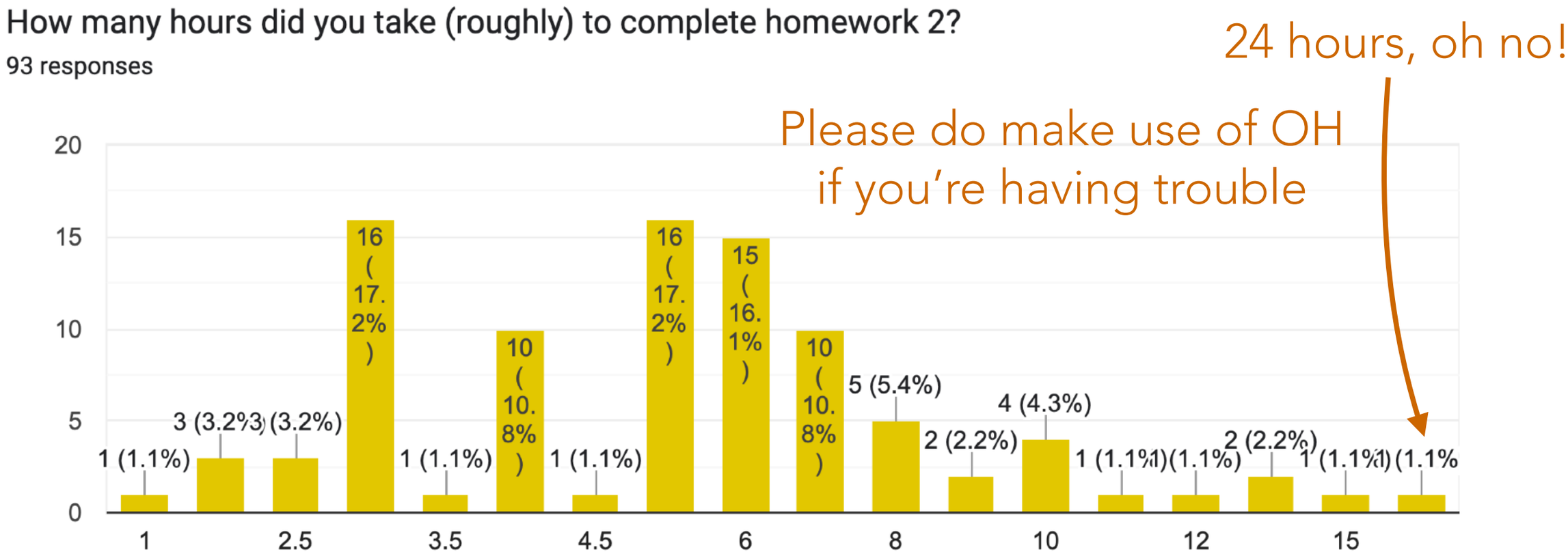
Carnegie Mellon University
Heinz College

95-865 Unstructured Data Analytics

Last lecture: Text generation with generative pre-trained transformers; a few more deep learning concepts; course wrap-up

Slides by George H. Chen

HW2 Questionnaire (1/2)



HW2 Questionnaire (2/2)

- Lots of students said that interpreting clusters or topics can be hard
  - Interpreting clusters or topics can indeed be challenging!
  - Even with newer topic models developed (such as BERTopic), interpretation can still be challenging depending on the dataset
- Lots of students said that t-SNE plots are confusing to interpret
  - Yes, this is indeed the case…
    - If you have some ground truth annotation that can be used to help color the data points, it might be easier seeing what's going on…
- A lot of students find StatQuest helpful
- A number of students said that they used Bilibili (I had no idea what this was until I looked it up)

Is this only in Chinese?

- A number of students expressed that it wasn't straightforward keeping track of what different sklearn models' fit/transform/etc do
  - This is good to write down on a cheatsheet including shape info for what comes out of transform/predict/predict_proba

(Flashback) Predict Next Character

> The opioid epidemic or opioid crisis is the rapid increase in the use of prescription and non-prescription opioid drugs in the United States and Canada in the 2010s.

Let's treat this string as a single data point (a time series of tokens)

For tokenization, let's split by individual characters
(so no need to use spaCy)

Given `['T']`, predict next character `'h'`

Given `['T', 'h']`, predict next character `'e'`

Given `['T', 'h', 'e']`, predict next character `' '`

Given `['T', 'h', 'e', ' ']`, predict next character `'o'`

…

If the string has $L + 1$ characters total, then there are L such prediction tasks

Reminder

Training point: `'The opi'`

Input to RNN language model: `'The op'`

Technically, the input is encoded as token IDs:

`[48, 60, 57, 1, 67, 68]`

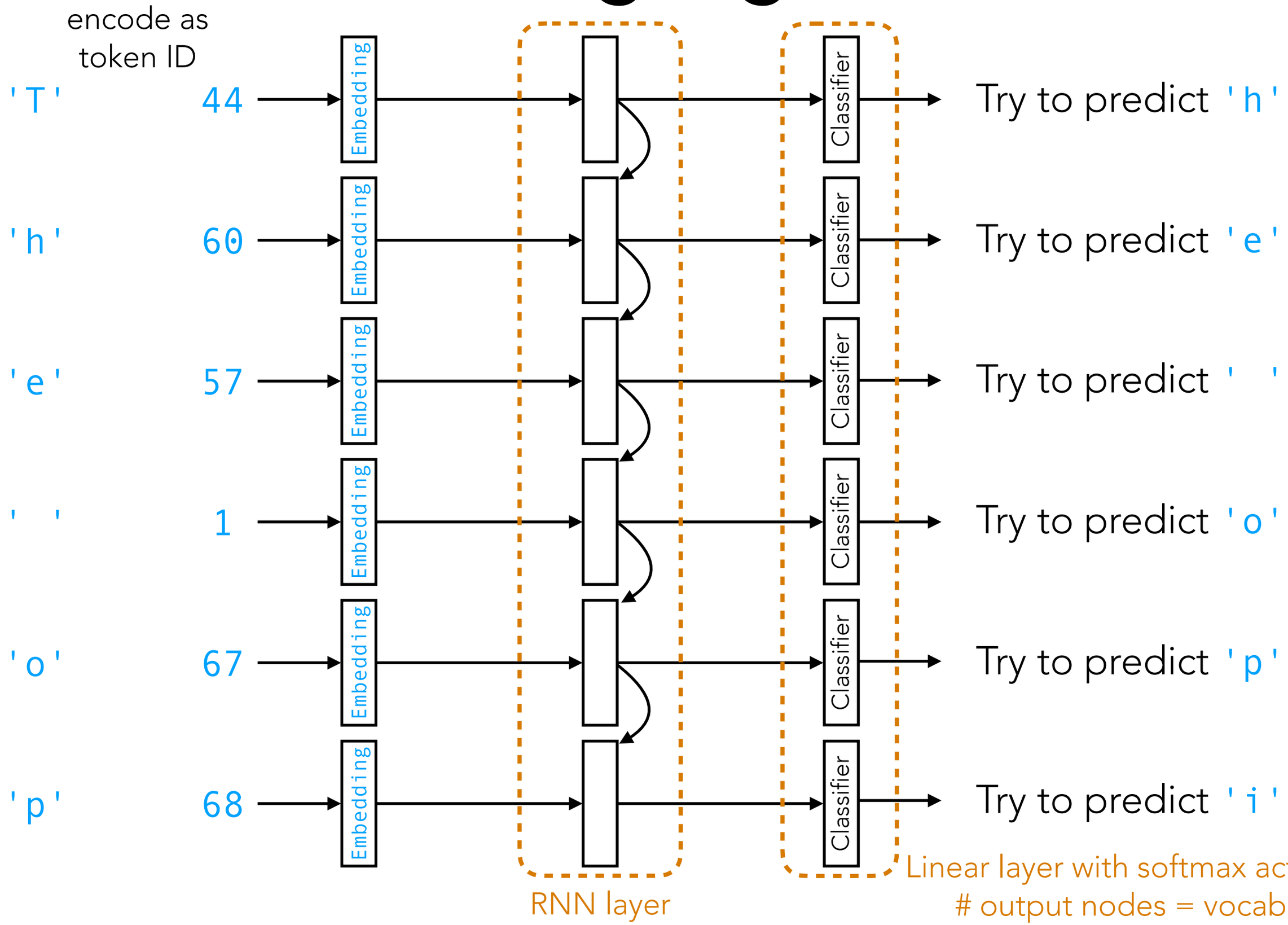
Desired output of RNN language model: `'he opi'`

Technically, the desired output is encoded as token IDs:

`[60, 57, 1, 67, 68, 61]`

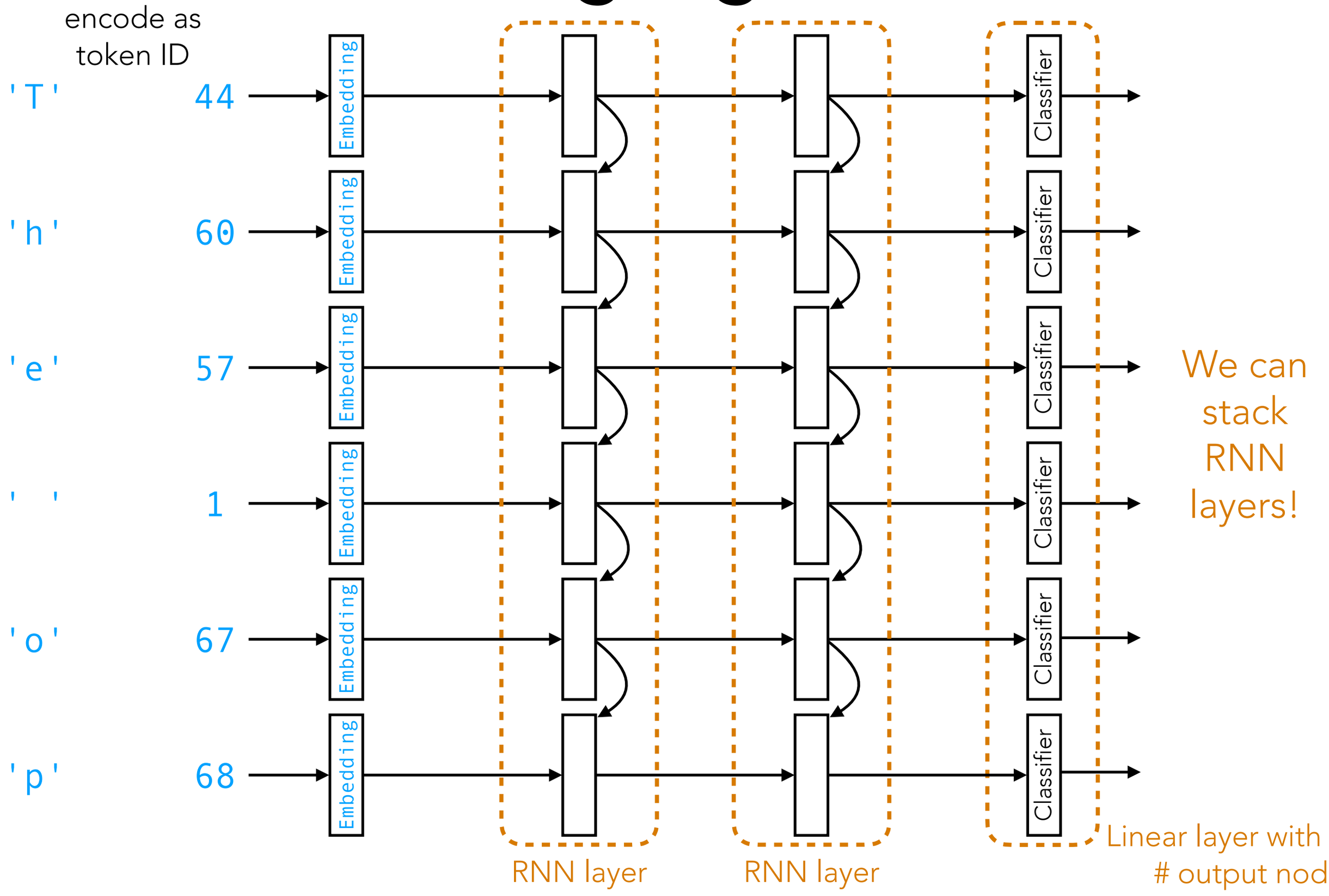
Train using minibatch gradient descent with cross entropy loss
(similar to other models we've seen in lecture)

RNN Language Model



Train using minibatch gradient descent with cross entropy loss
(similar to other models we've seen in lecture)

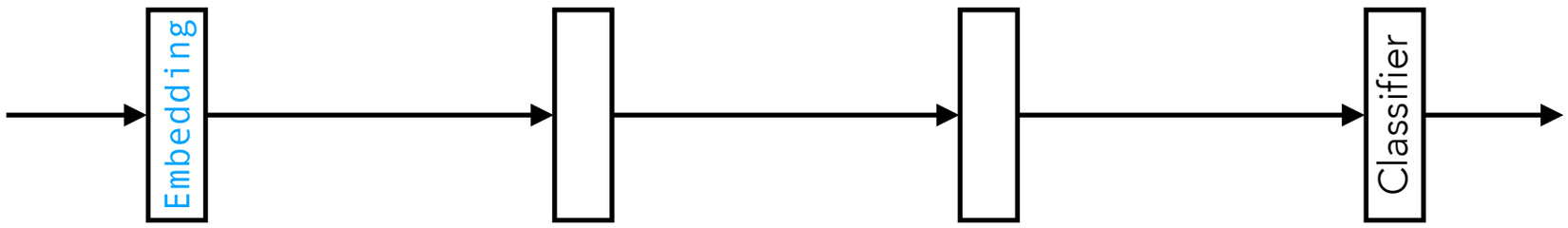
RNN Language Model



How to Generate Text After Model Training

encode as
token ID

'T' 44



start with some
initial character

output is a probability
distribution over characters

randomly sample from this distribution
to generate next character!

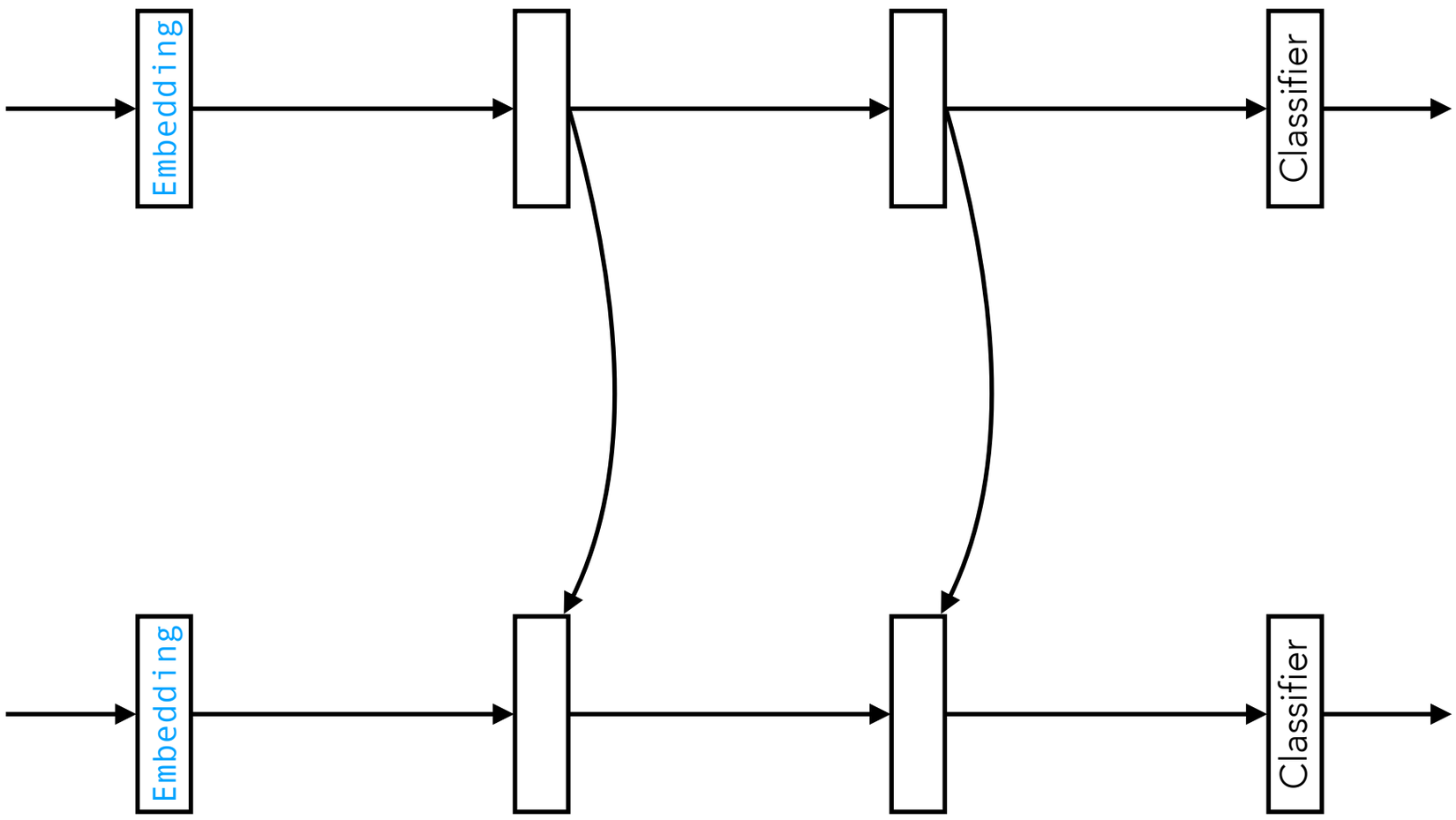
'e'

How to Generate Text After Model Training

encode as
token ID

'T' 44

start with some
initial character



'e' 57

output is a probability
distribution over characters

randomly sample from this distribution
to generate next character!

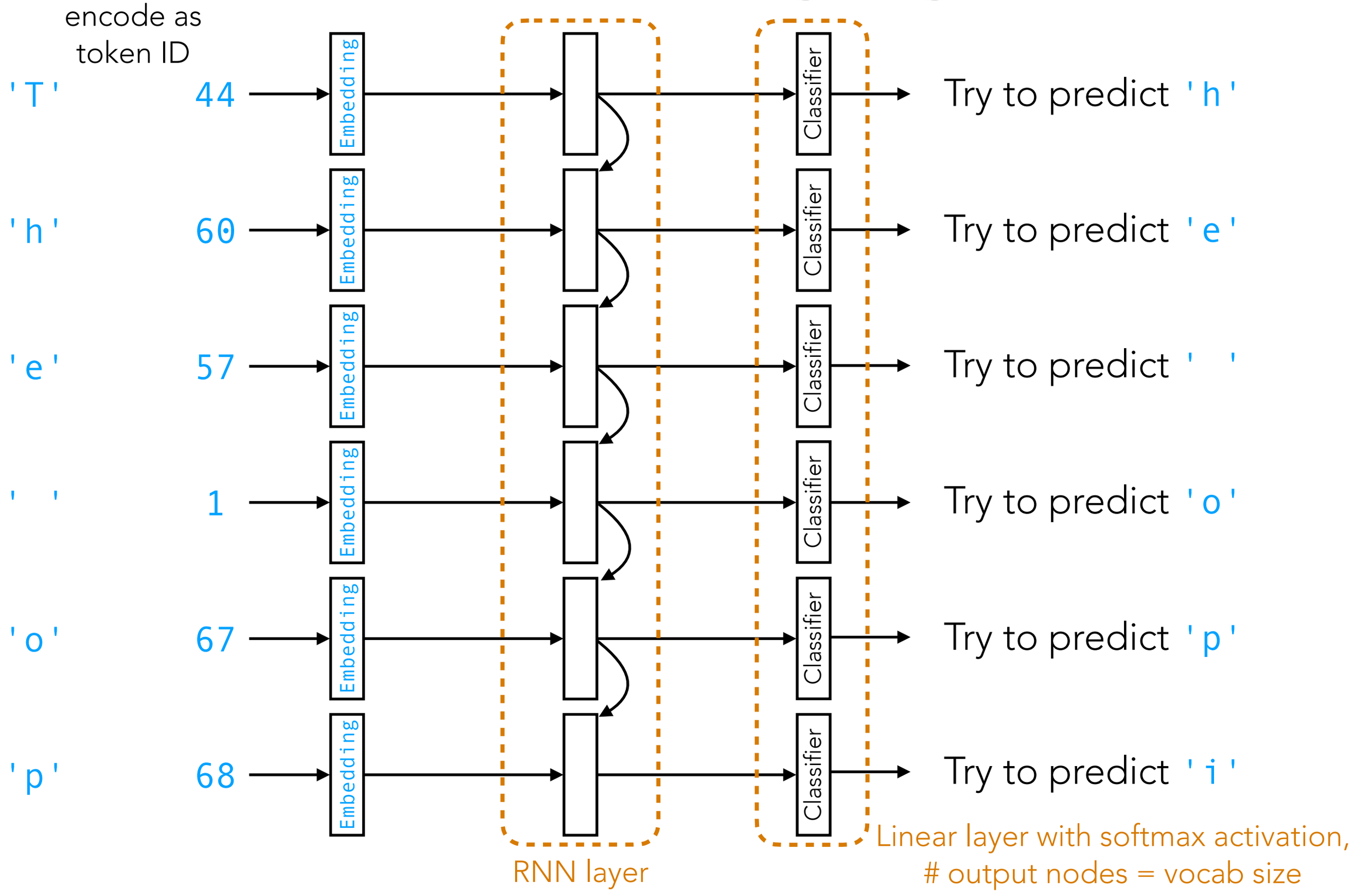
's'

Keep generating text in this manner!

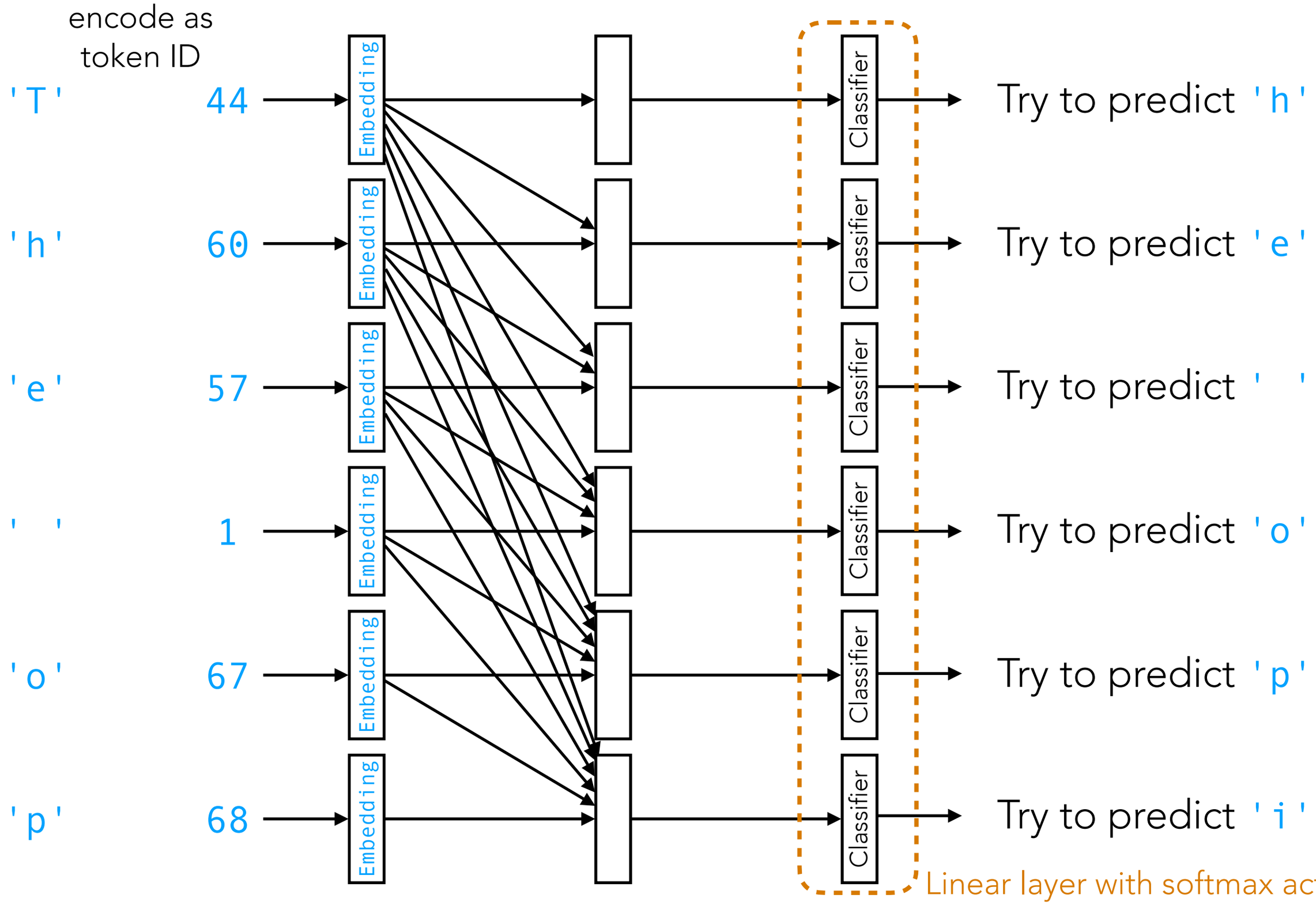
An Alternative Solution: Generative Pre-trained Transformers (GPTs)

Explicitly figure out how to weight the contribution of the current & past time steps

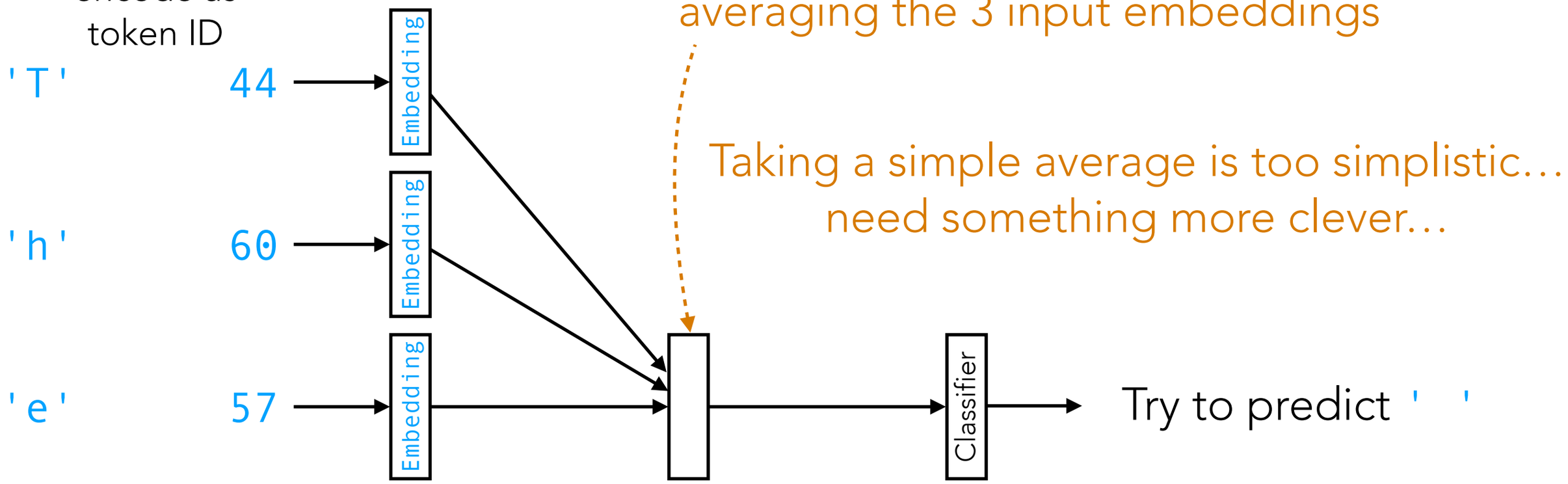
(Flashback) RNN Language Model



This sort of dependence is "causal": any time step can only depend on its current input and all past inputs (and **not** on future time steps' inputs)



Let's focus on time step 2's prediction task for the moment…



Bad idea: have this box correspond to averaging the 3 input embeddings

Taking a simple average is too simplistic… need something more clever…

How should we combine information from the input embeddings?

Another issue: the input embeddings by themselves do not contain information about *when* the time steps happened

→ **Let's address this issue first**

encode as
token ID

'T' 44

Position: 0

'h' 60

Position: 1

'e' 57

Position: 2

Embedding

Embedding

Embedding

Classifier

Try to predict ' '

Each embedding: 1D table with D entries

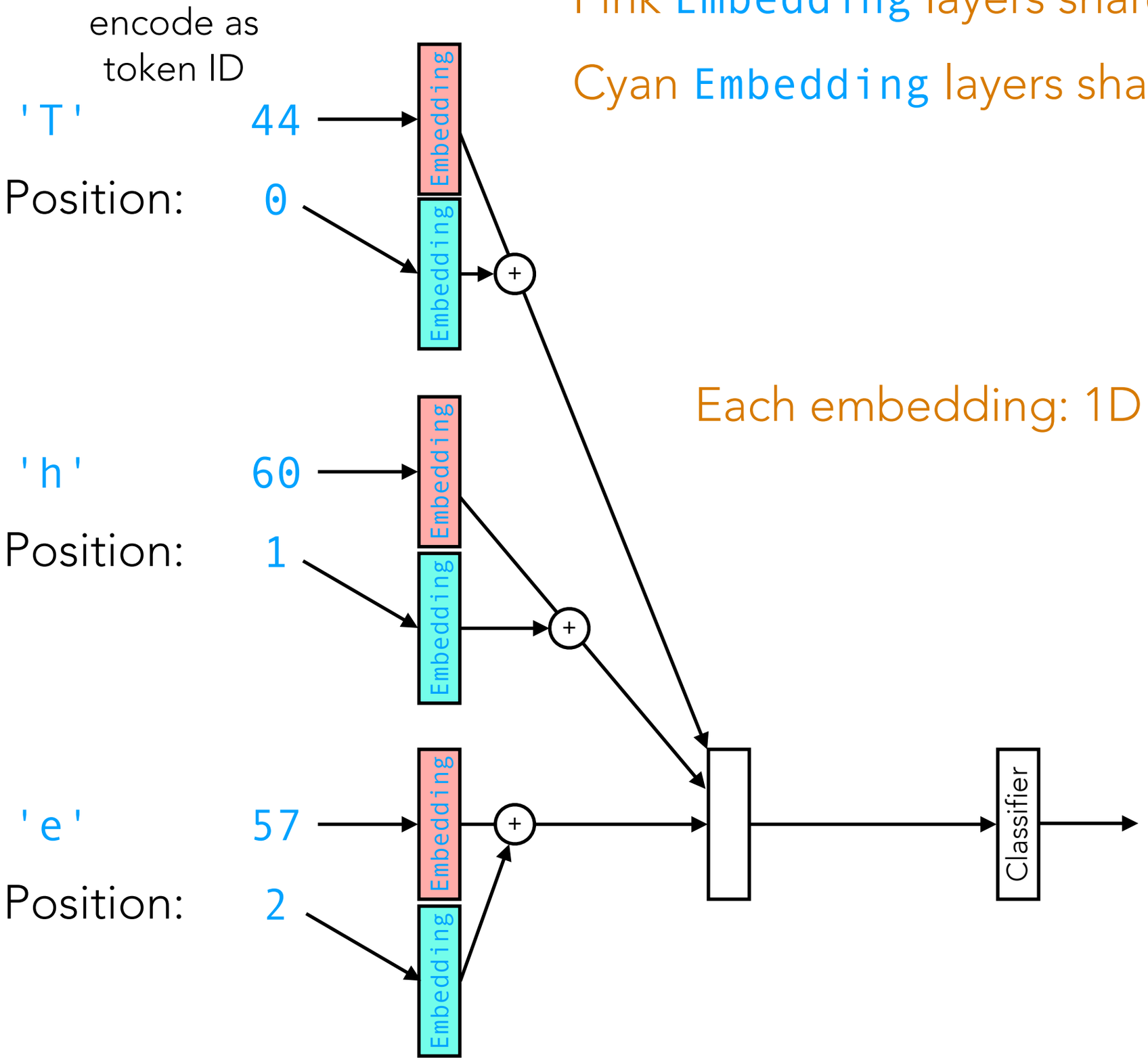
a hyperparameter
(512 in the demo)

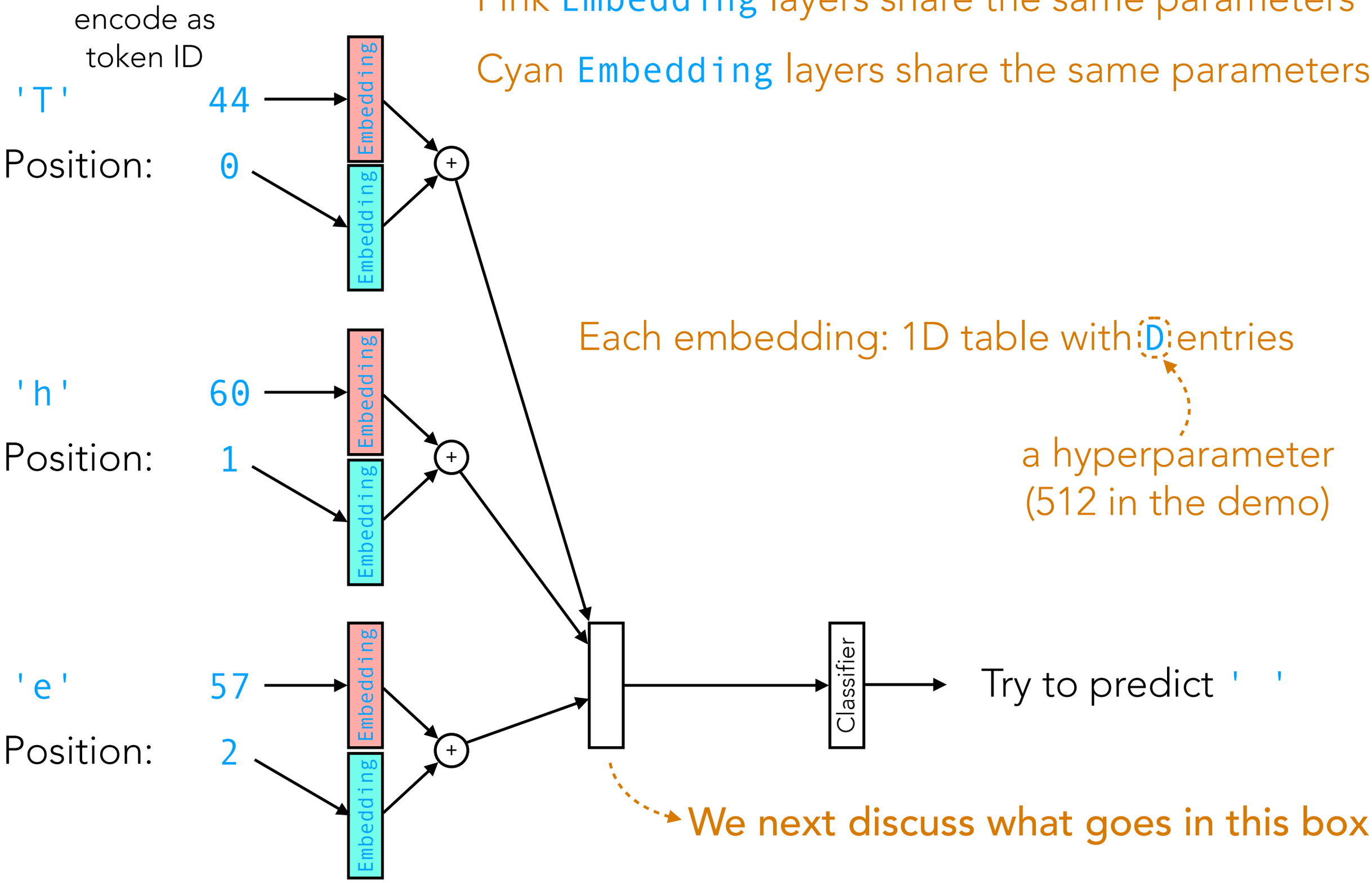
Pink `Embedding` layers share the same parameters

Cyan `Embedding` layers share the same parameters

Each embedding: 1D table with `D` entries

a hyperparameter
(512 in the demo)

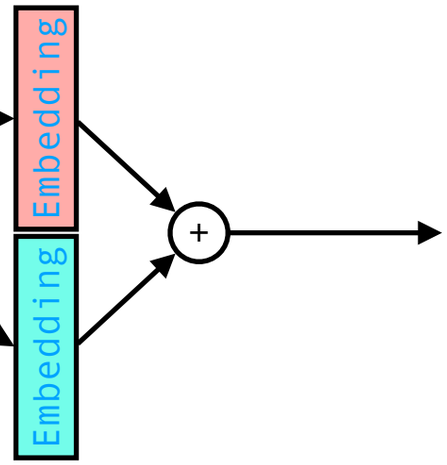



encode as
token ID
'T'
44
Position:
0
'h'
60
Position:
1
'e'
57
Position:
2
Embedding
Embedding
Embedding
Embedding
Embedding
Embedding
+
+
+
Classifier
Try to predict ' '
Pink Embedding layers share the same parameters
Cyan Embedding layers share the same parameters
Each embedding: 1D table with D entries
a hyperparameter
(512 in the demo)
We next discuss what goes in this box

encode as
token ID

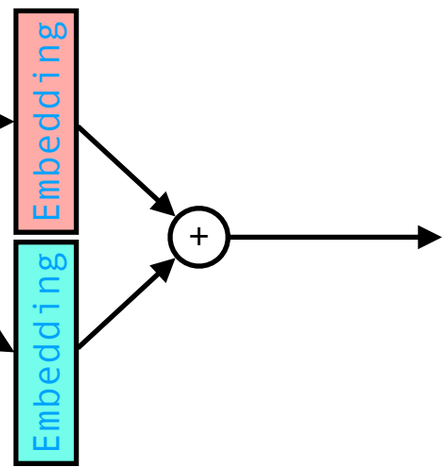
'T' 44

Position: 0



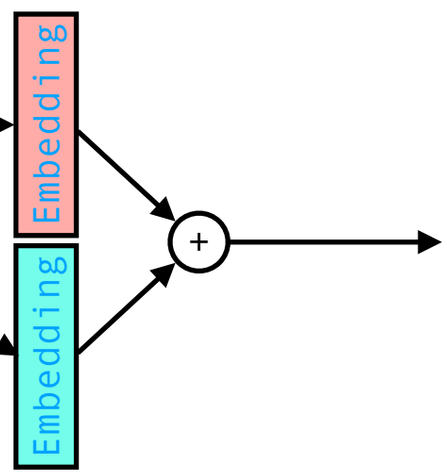
'h' 60

Position: 1



'e' 57

Position: 2



encode as token ID

'T' 44

Position: 0

Embedding

Embedding

+

Linear → key → key[0] "Here's how to find me"

Linear → query → query[0] "Here's what I'm looking for"

Linear → value → value[0] "Here's my information"

'h' 60

Position: 1

Embedding

Embedding

+

Linear → key → key[1]

Linear → query → query[1]

Linear → value → value[1]

'e' 57

Position: 2

Embedding

Embedding

+

Linear → key → key[2]

Linear → query → query[2]

Linear → value → value[2]

keys/queries/values are 1D tables with the same # entries: H

a hyperparameter (256 in the demo)

Remember: at this point, we are only computing the output for time step 2

How much should time step 0's information contribute (to the output for time step 2)?

Idea: make the contribution amount dependent on:

```
w[0] = np.dot(query[2], key[0])
```

How much should time step 1's information contribute?

```
w[1] = np.dot(query[2], key[1])
```

How much should time step 2's information contribute?

```
w[2] = np.dot(query[2], key[2])
```

Let's normalize the weights so they are probabilities:

```
w_norm = softmax(w)
```

Output at time step 2:

```
w_norm[0]*value[0] + w_norm[1]*value[1] \
+ w_norm[2]*value[2]
```

Remember: at this point, we are only computing the output for time step 2

How much should time step 0's information contribute (to the output for time step 2)?

Idea: make the contribution amount dependent on:

```
w[0] = np.dot(query[2], key[0])
```

How much should time step 1's information contribute?

```
w[1] = np.dot(query[2], key[1])
```

How much should time step 2's information contribute?

```
w[2] = np.dot(query[2], key[2])
```

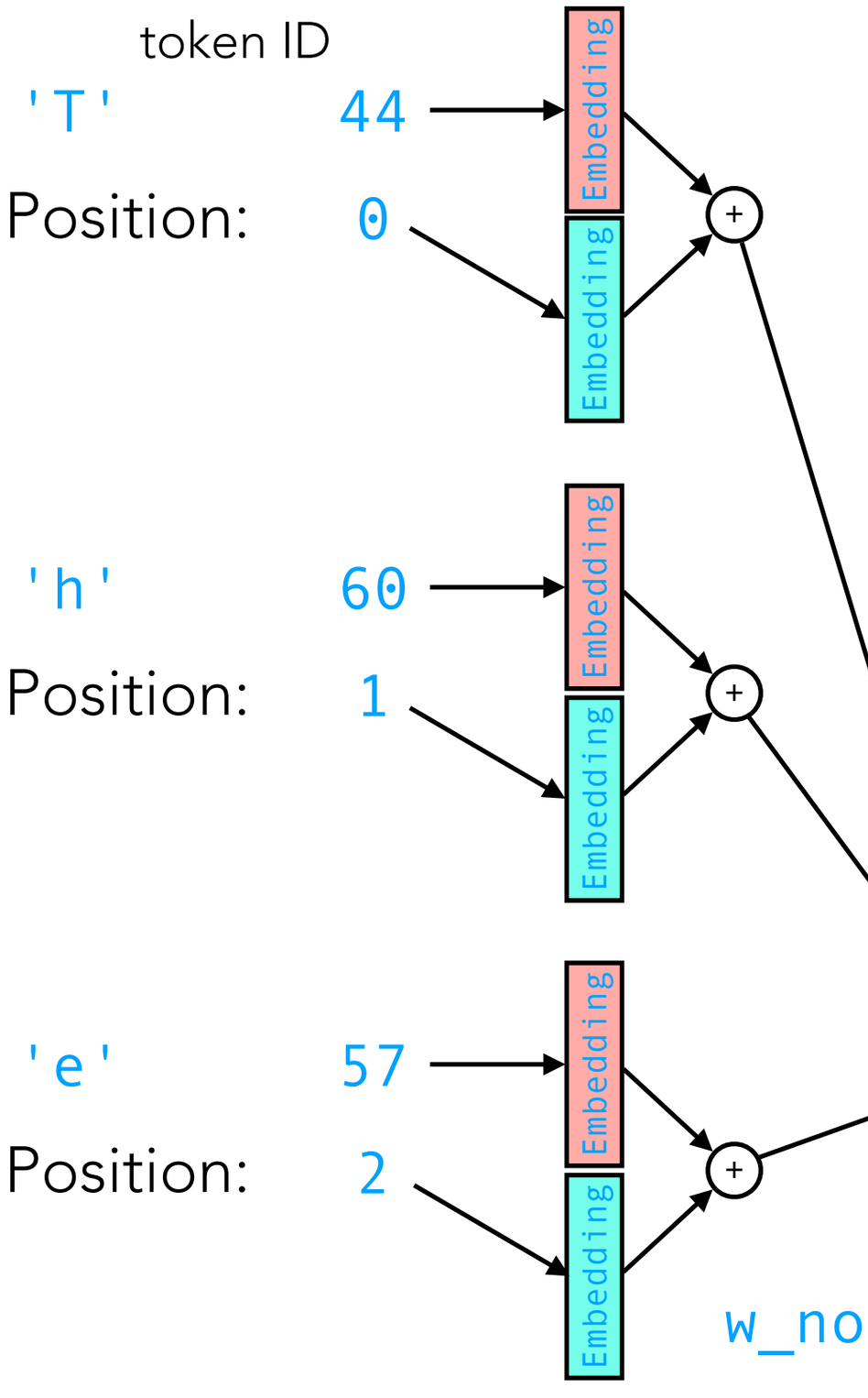
Let's normalize the weights so they are probabilities:

```
w_norm = softmax(w / np.sqrt(H))
```

In practice: include this division (helps with training)

Output at time step 2:

```
w_norm[0]*value[0] + w_norm[1]*value[1] \
    + w_norm[2]*value[2]
```



The hope: keys, queries, and values that get learned help with prediction

But what if we get unlucky and the keys, queries, and values found aren't great (or only focus on a single concept)?

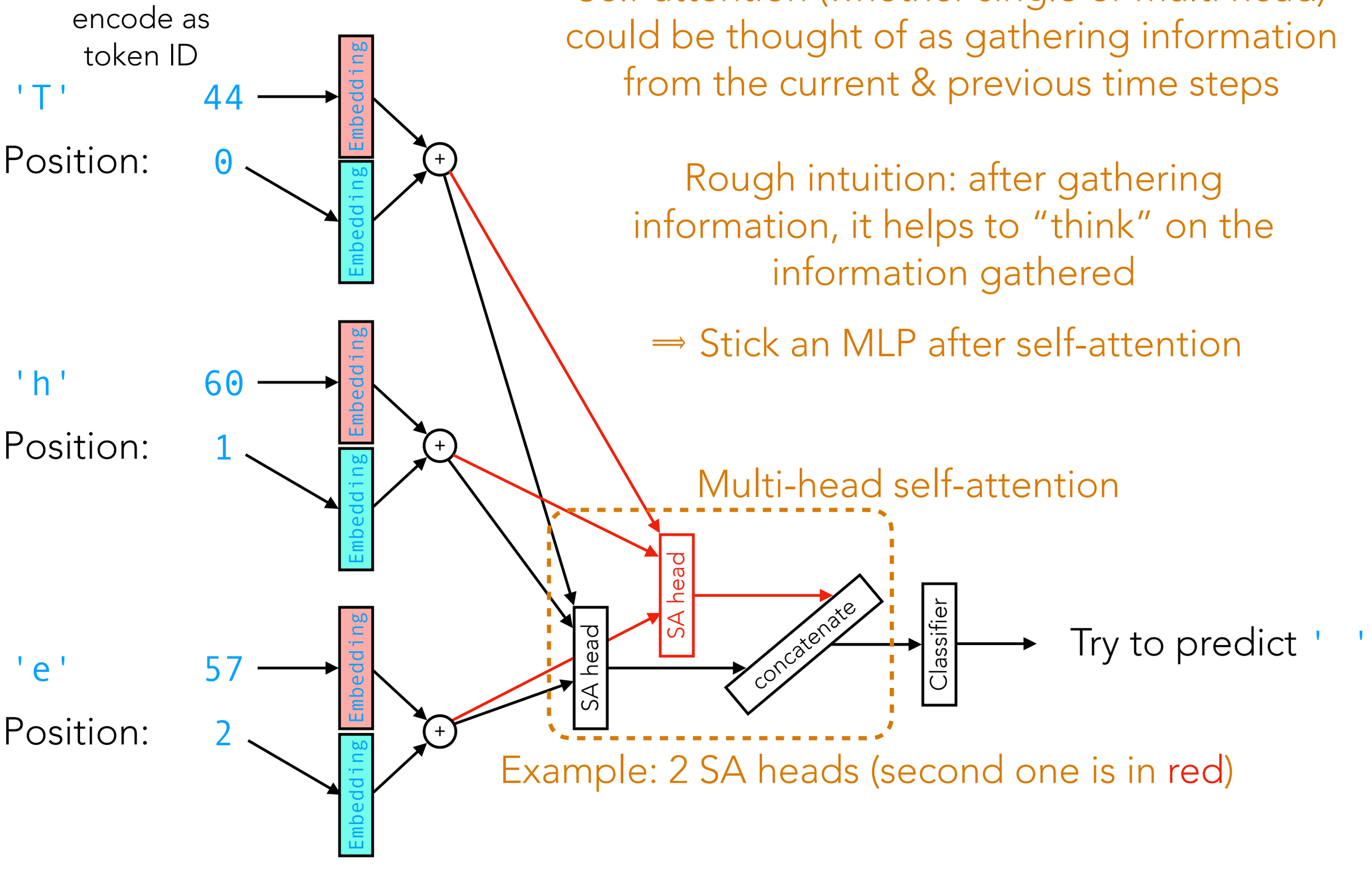
Analogy: imagine if we used a Conv2d layer but only used 1 filter and hoped that the 1 filter captures everything

The fix: use many self-attention heads

Self-attention (whether single or multi head) could be thought of as gathering information from the current & previous time steps

Rough intuition: after gathering information, it helps to "think" on the information gathered

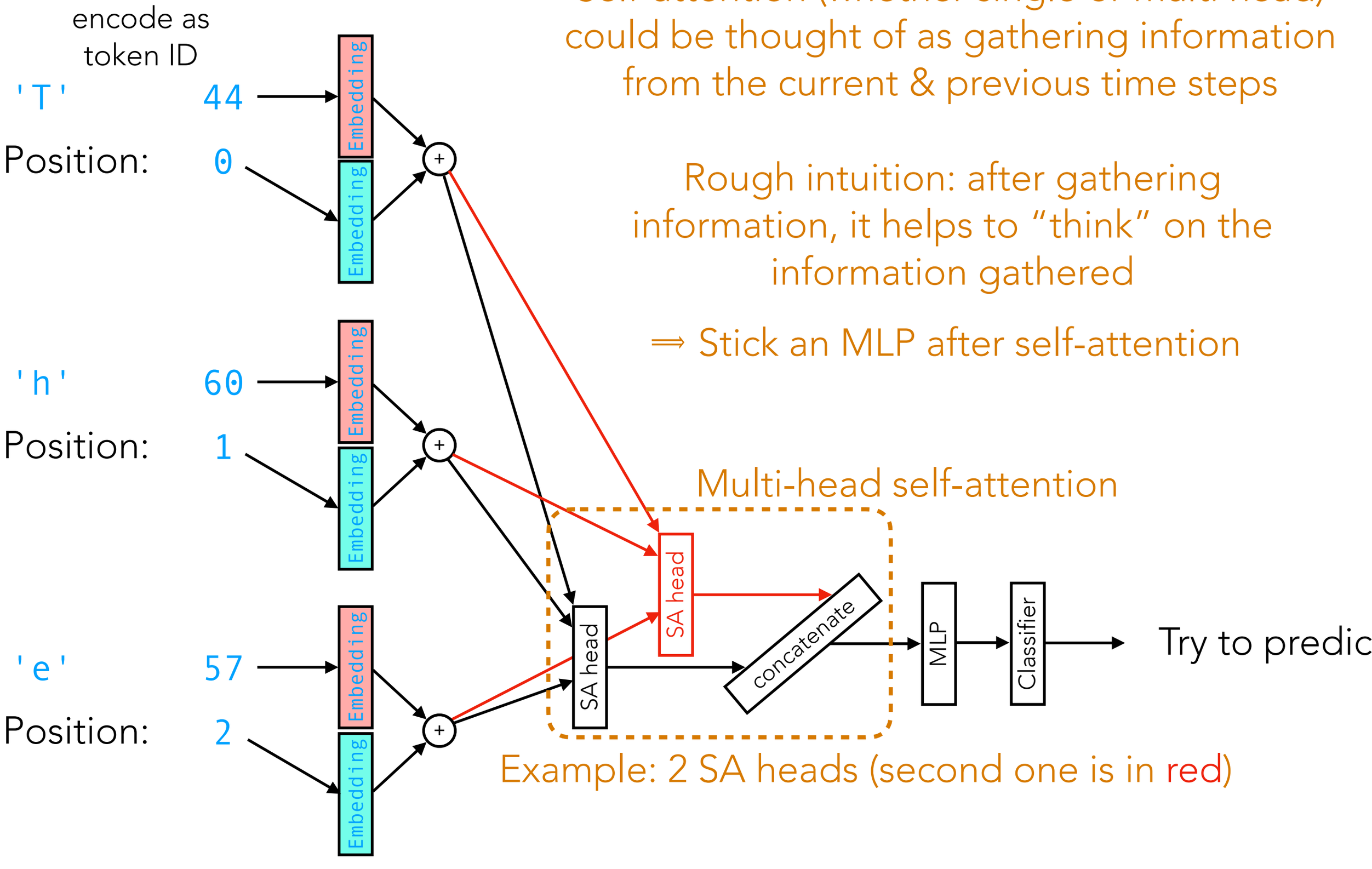
⟹ Stick an MLP after self-attention



Self-attention (whether single or multi head) could be thought of as gathering information from the current & previous time steps

Rough intuition: after gathering information, it helps to "think" on the information gathered

⟹ Stick an MLP after self-attention



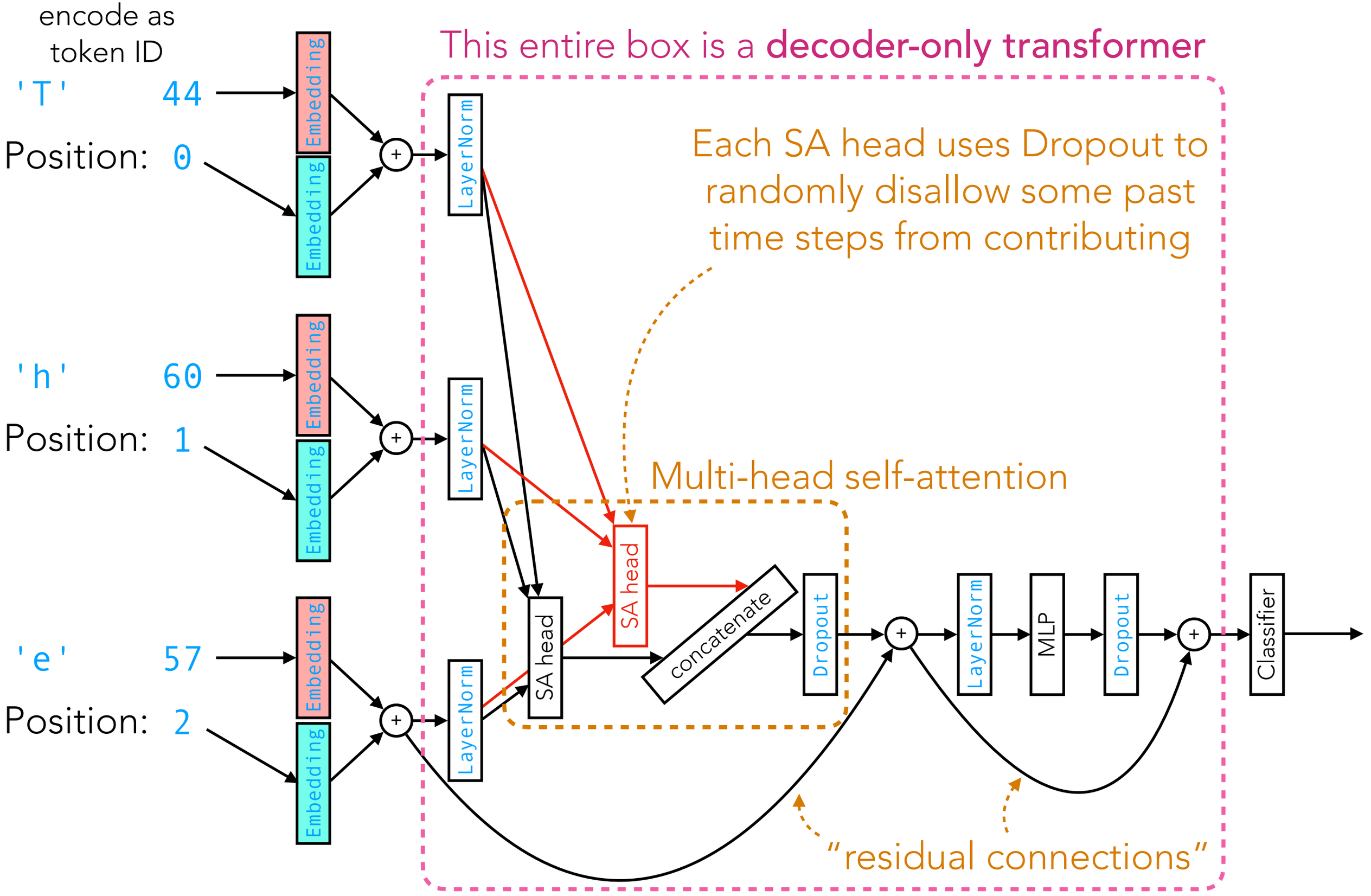
There are a few implementation details that I won't go over in lecture

Basically, it turns out that when neural nets get very deep, training can be more difficult without some now-standard tricks (these tricks work with *many* neural net architectures, not just GPTs)

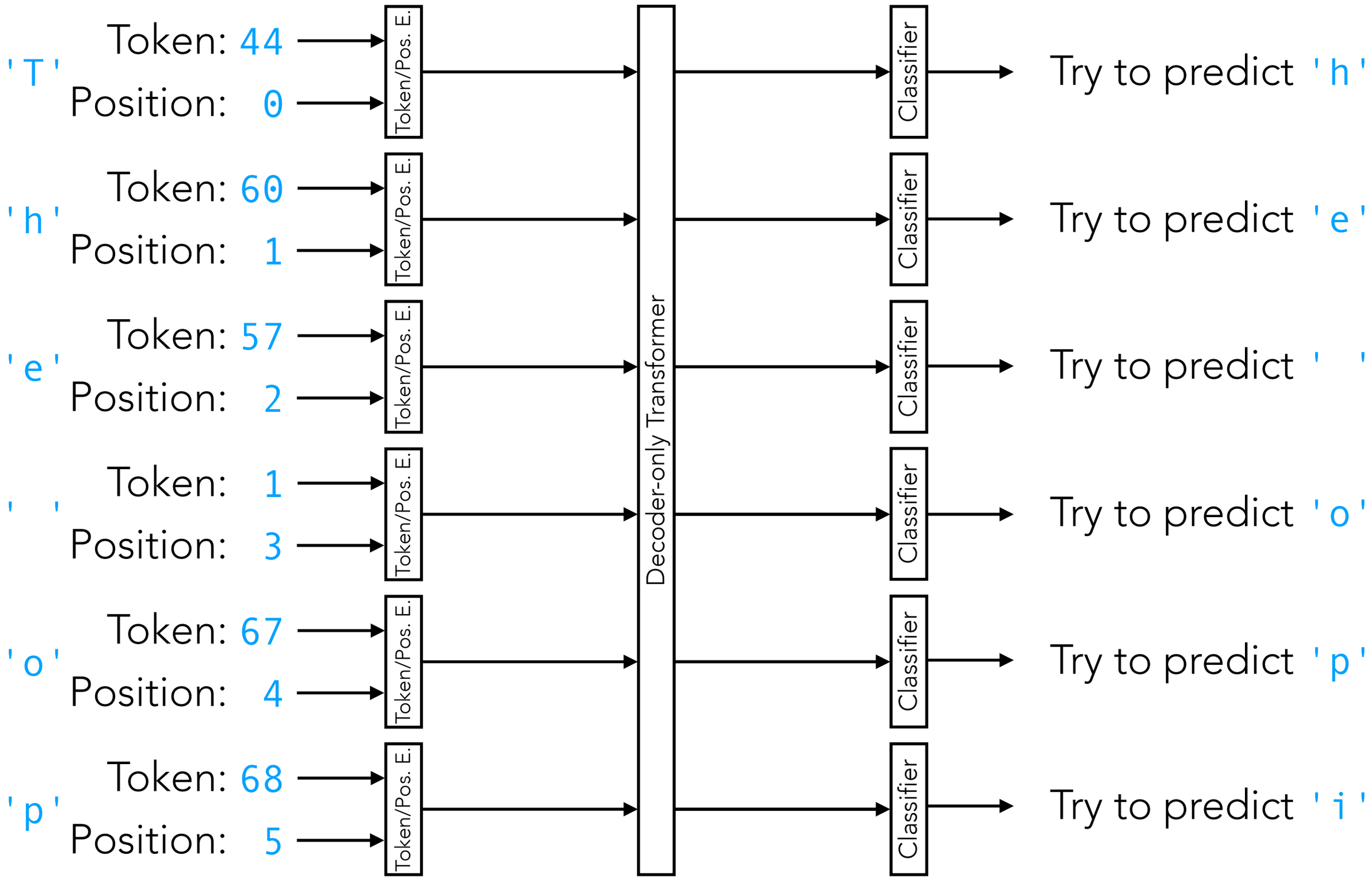
- LayerNorm
- Residual connections
- Dropout

You're not expected to know these technical details

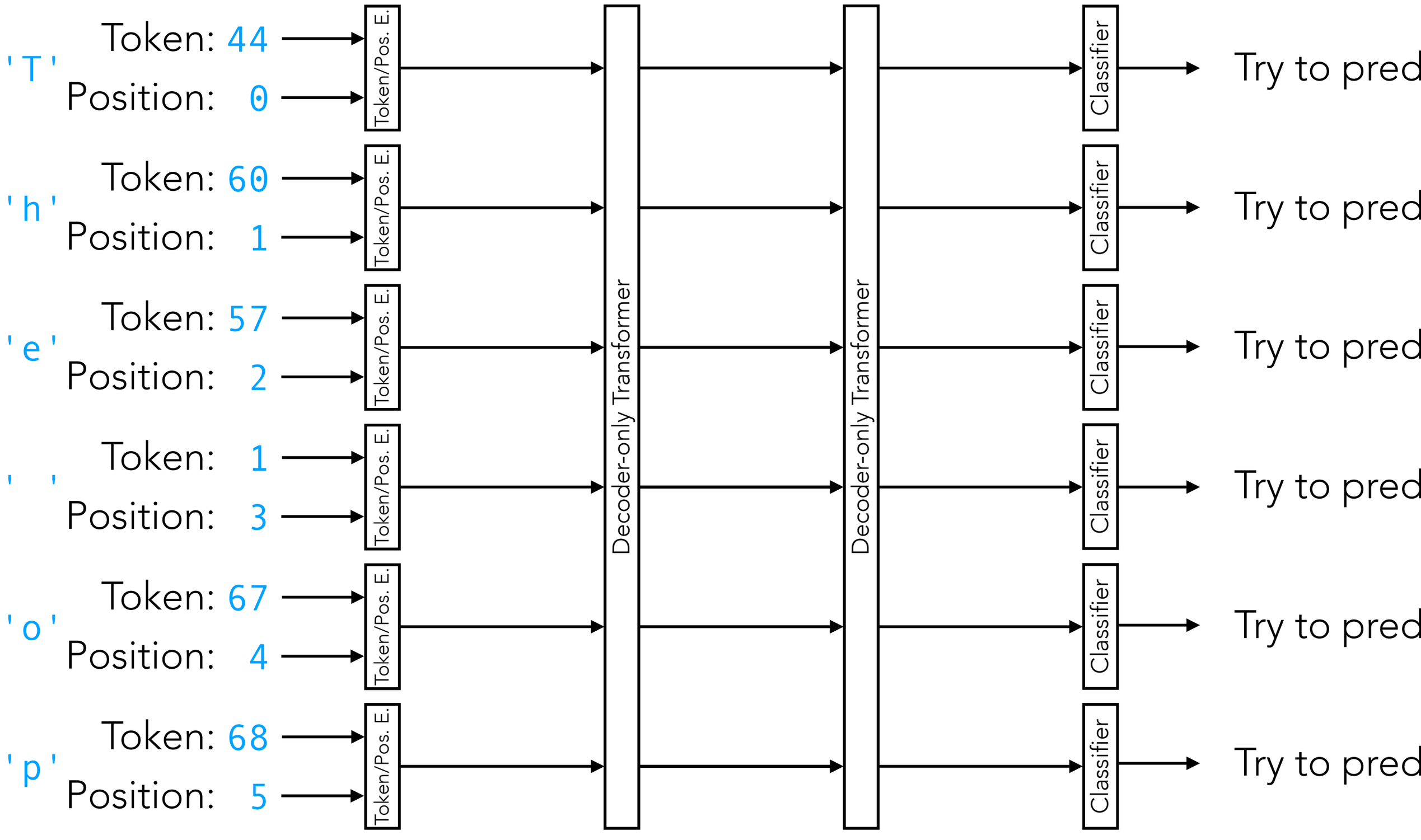
Also, there are some standard strategies for initializing GPT training


encode as token ID
'T' 44
Position: 0
'h' 60
Position: 1
'e' 57
Position: 2
Embedding
Embedding
Embedding
Embedding
Embedding
Embedding
This entire box is a decoder-only transformer
Each SA head uses Dropout to randomly disallow some past time steps from contributing
LayerNorm
LayerNorm
LayerNorm
Multi-head self-attention
SA head
SA head
concatenate
Dropout
LayerNorm
MLP
Dropout
Classifier
"residual connections"

Generative Pre-trained Transformer (GPT)

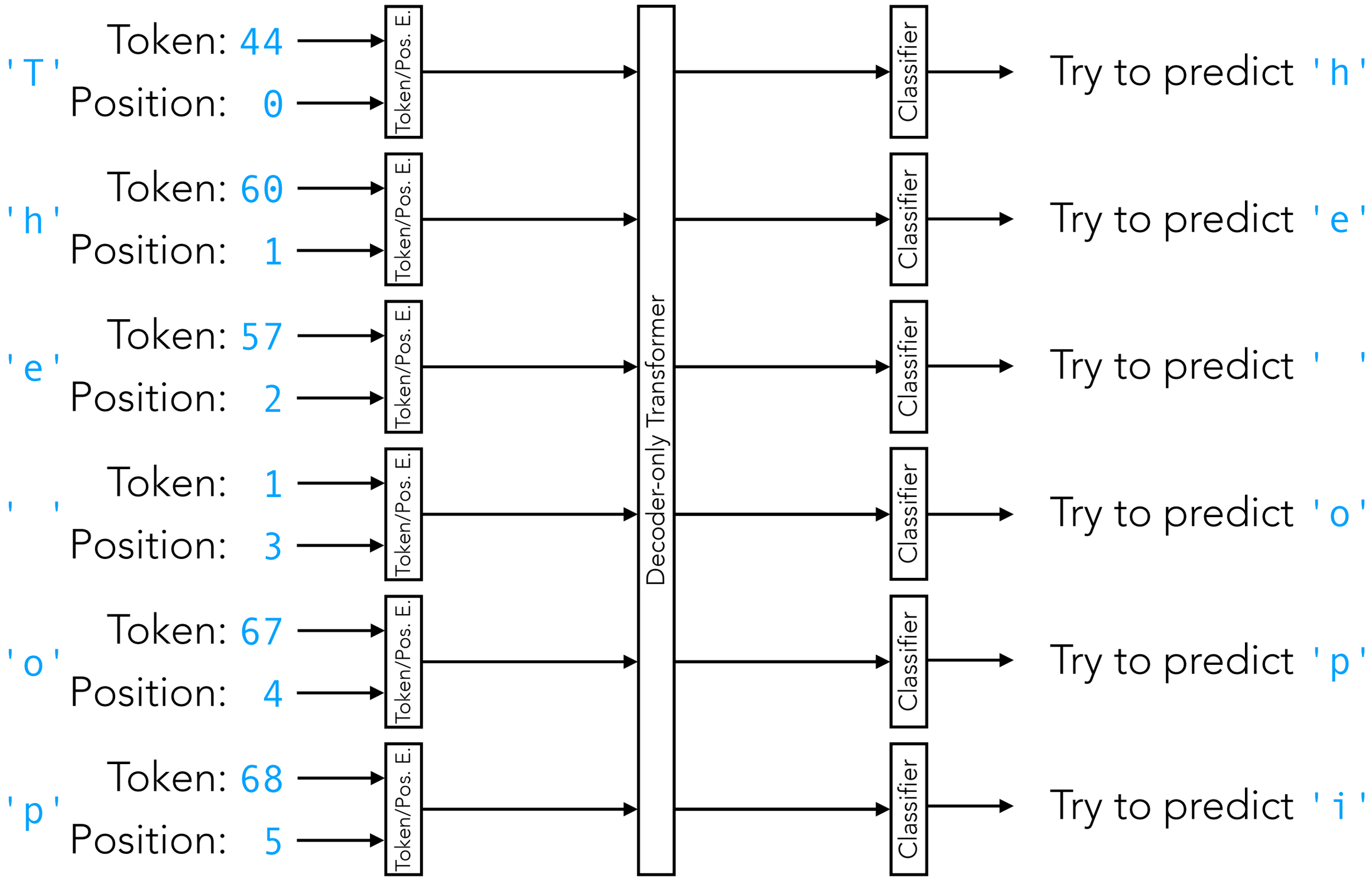


Generative Pre-trained Transformer (GPT)

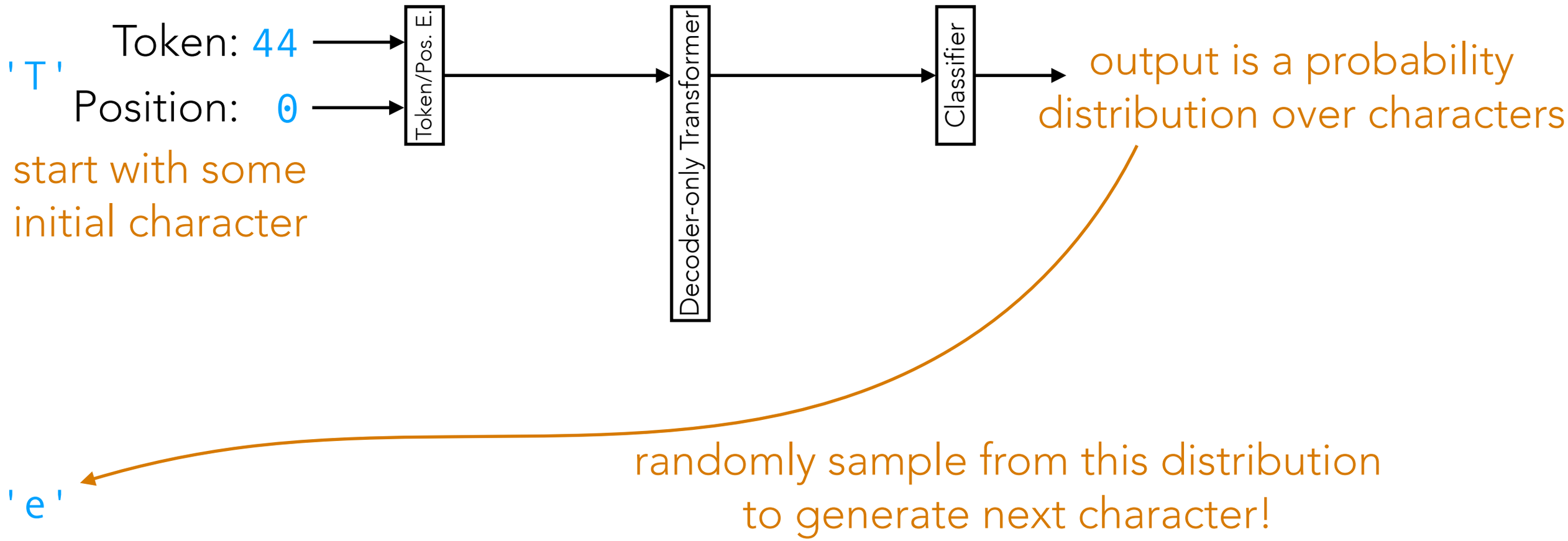


It is possible to stack transformer layers (similar to RNN layers)

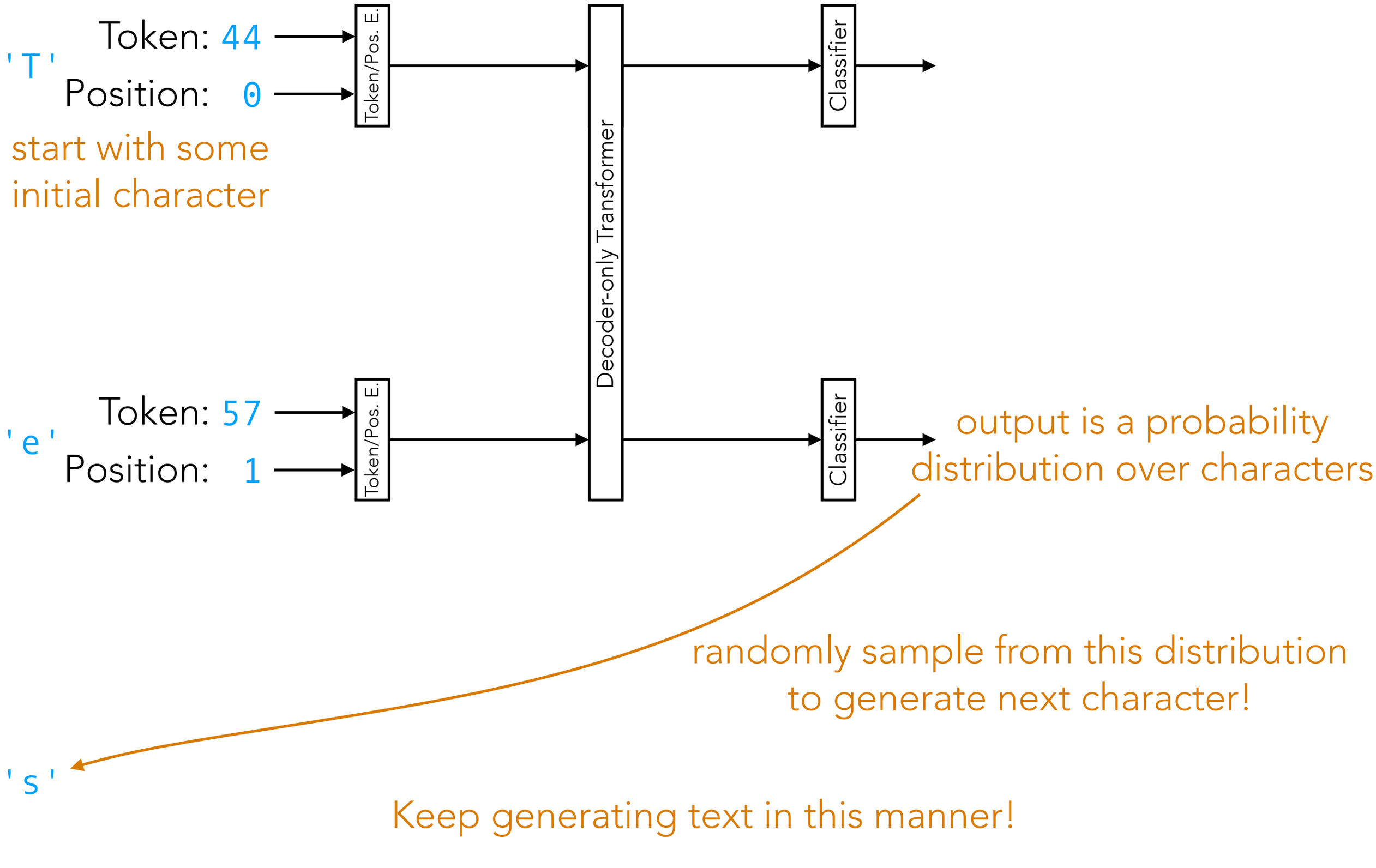
Generative Pre-trained Transformer (GPT)



How to Generate Text After Model Training



How to Generate Text After Model Training



GPT

Demo

How to Get GPTs to Answer Prompts

A system like ChatGPT is trained in two phases

- First, it is "pre-trained" on a massive chunk of the internet using the prediction task we described already (this prediction task does *not* require any human annotations)

After this pre-training step, the model can randomly generate text but doesn't know how to answer prompts yet (the model is "unaligned" with human goals at this point)

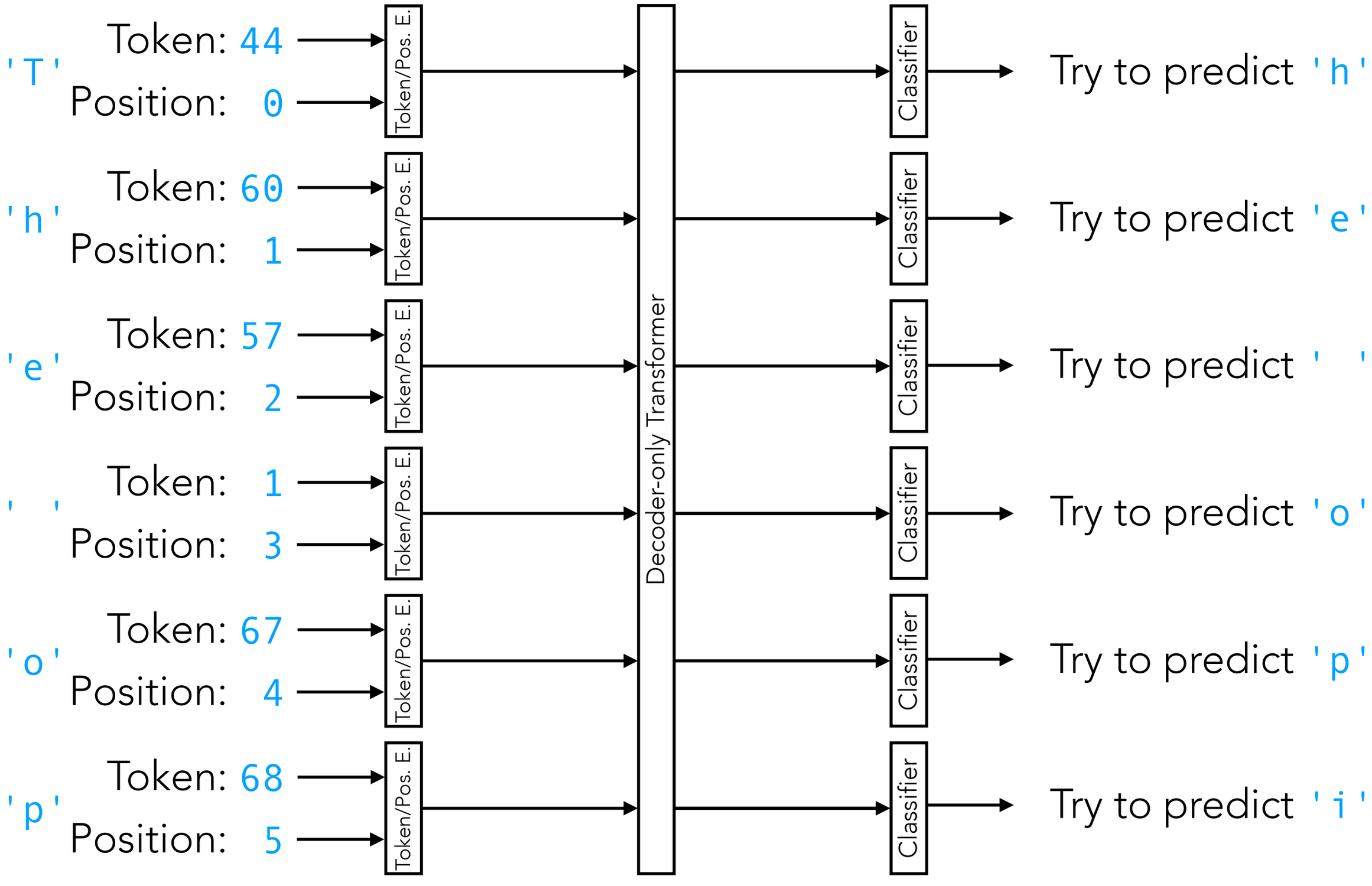
- Next, we "fine-tune" the model by giving it labeled training data showing questions & answers, and over time, we improve the model by letting humans scoring responses of the model

This is called "reinforcement learning with human feedback" (RLHF)

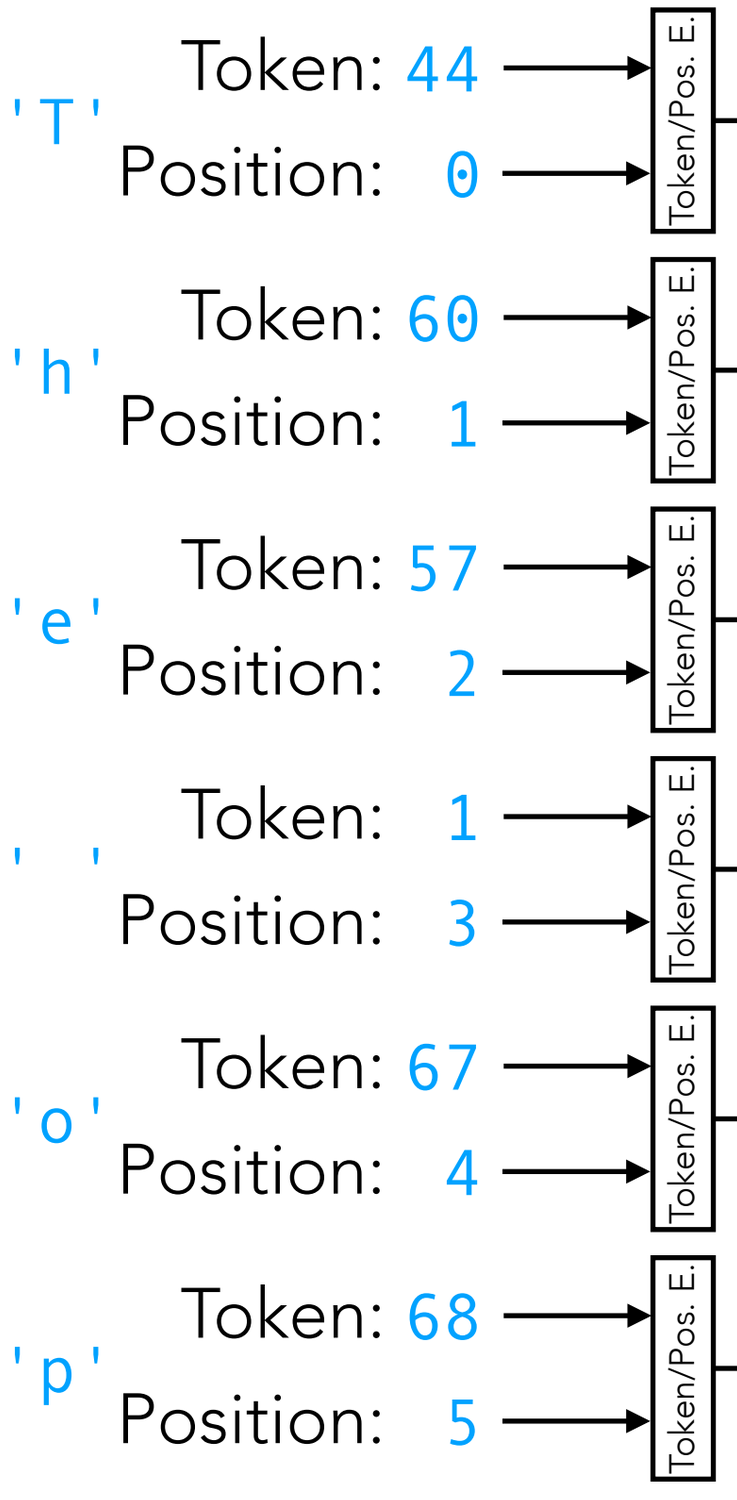
We focused on this first step today

I'll briefly discuss fine-tuning (but not reinforcement learning)

(Flashback) Generative Pre-trained Transformer (GPT)



(Flashback) Generative Pre-trained Transformer (GPT)

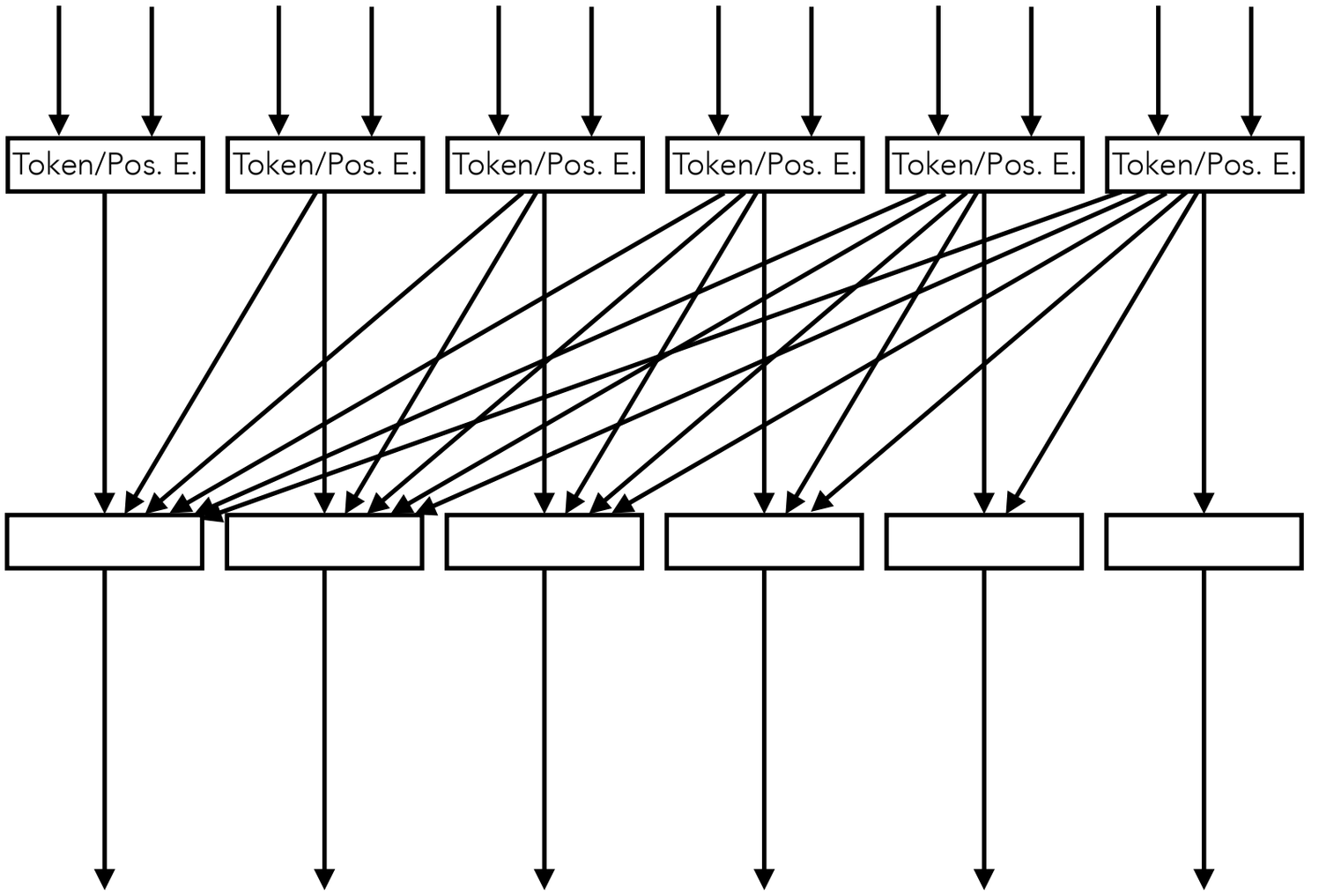


The output before the classifier could be thought of as a token embedding that accounts for context

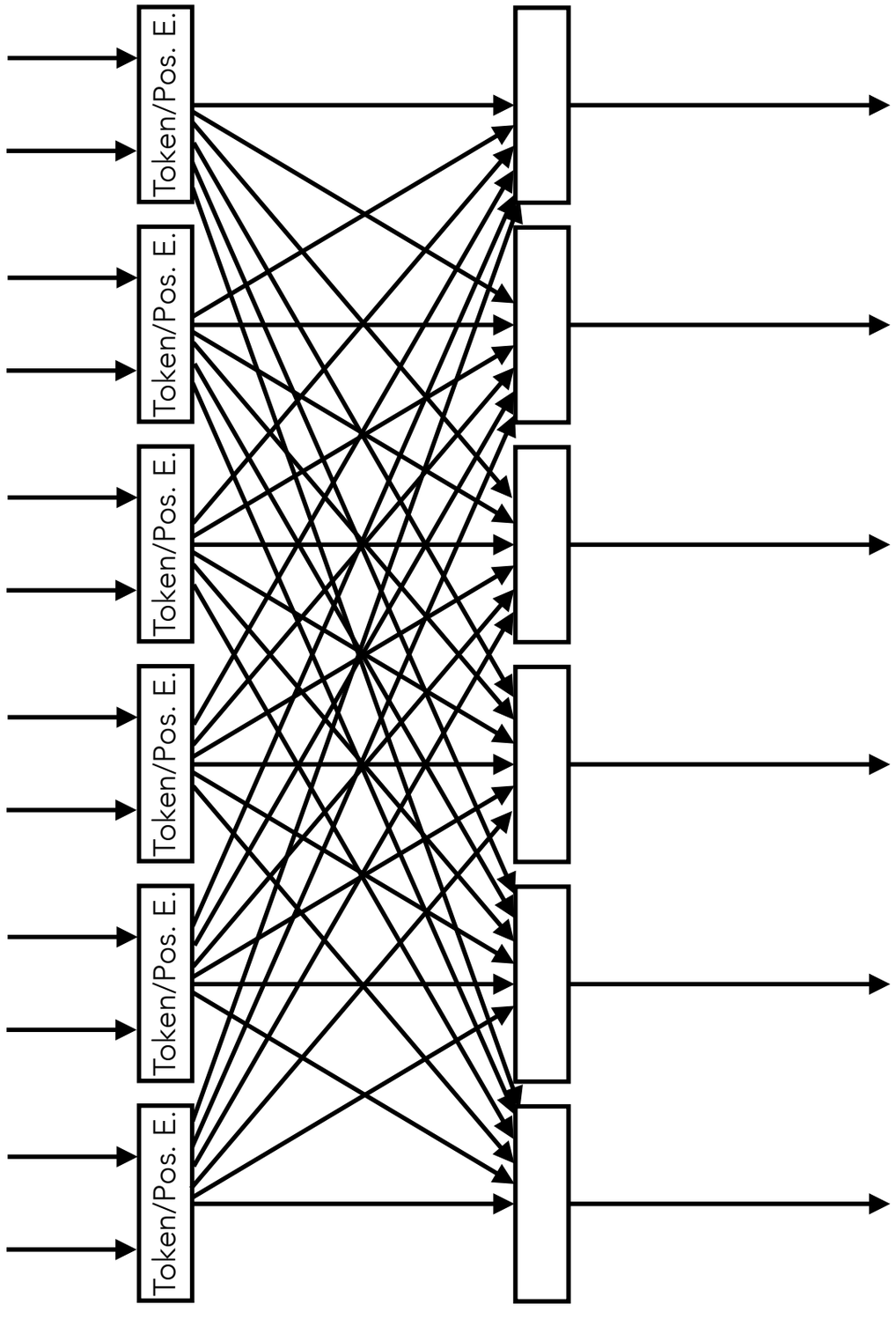
If we tokenized instead using words, then we would have word embeddings

The BERT model is basically what we showed today except **without the causal constraint** & **with a different tokenizer**

(As we saw previously, BERT's tokenization uses words and sometimes subwords)


Token/Pos. E.
Token/Pos. E.
Token/Pos. E.
Token/Pos. E.
Token/Pos. E.
Token/Pos. E.
causal dependence

BERT (2018)



no causal dependence

The prediction at any time step depends on the input at all time steps

This lack of causal dependence is also sometimes referred to as "bidirectional"

A transformer layer like this without a causal constraint is sometimes called an "encoder-only" transformer layer

BERT is short for Bidirectional Encoder Representations from Transformers

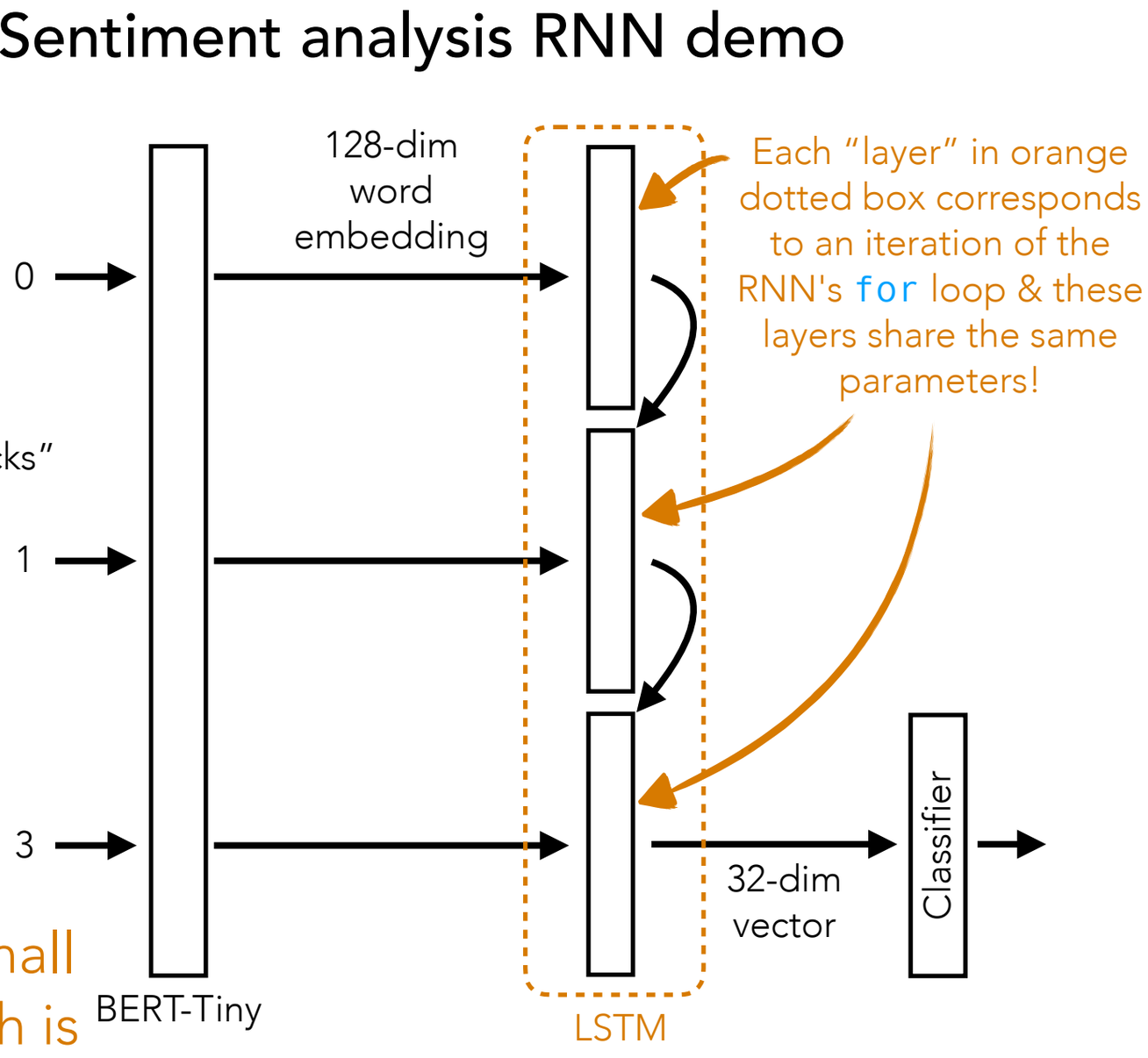
Fine-Tuning

Load in an already trained model, possibly change the last few layers, and modify it for our purposes

We loaded in a pre-trained BERT-Tiny model, which is a compressed version of BERT-Large, trained on a large dataset including BooksCorpus (800M words) + English Wikipedia (2500M words)

We then fine-tuned BERT-Tiny for our sentiment analysis neural net

Note that we fine-tuned on a relatively small dataset (only 25000 training reviews, which is much smaller than BooksCorpus/English Wikipedia)



Handling Small Datasets

- Fine-tuning has an extremely important application: it allows us to use an existing model trained on a *massive* dataset to help us with a new prediction task where we might only have a *small* dataset

We just talked about this for the sentiment analysis demo (previous slide)

ChatGPT/GPT 4.0:
GPT pre-trained on massive dataset (exact size undisclosed…)
Fine-tune on human-annotated training dataset (of Q&A pairs and scores of how good the system's automatically generated Q&A pairs are), known to be much smaller than what the model is pre-trained on

Handling Small Datasets

- Fine-tuning has an extremely important application: it allows us to use an existing model trained on a *massive* dataset to help us with a new prediction task where we might only have a *small* dataset
- Another extremely important strategy: **data augmentation (randomly perturb training data to get more training data)**

Training image

Training label: cat

Mirrored

Still a cat!

Rotated & translated

Still a cat!

We just turned 1 training example in 3 training examples!

State-of-the-art vision systems are all trained with data augmentation!

Allowable perturbations depend on data

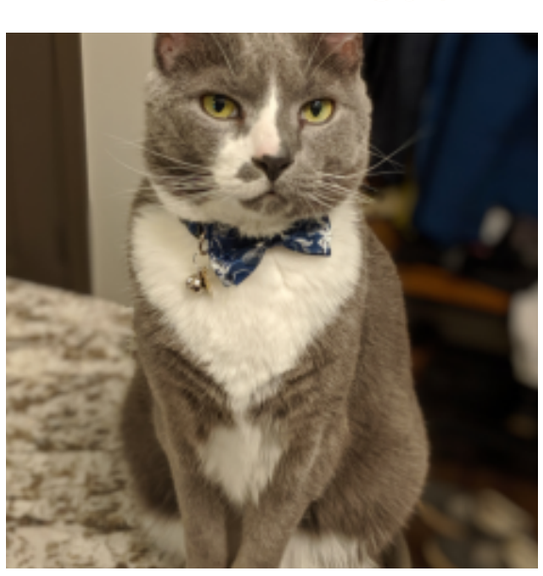
(e.g., for handwritten digits, rotating a 6 or 9 by 180 degrees would be bad)

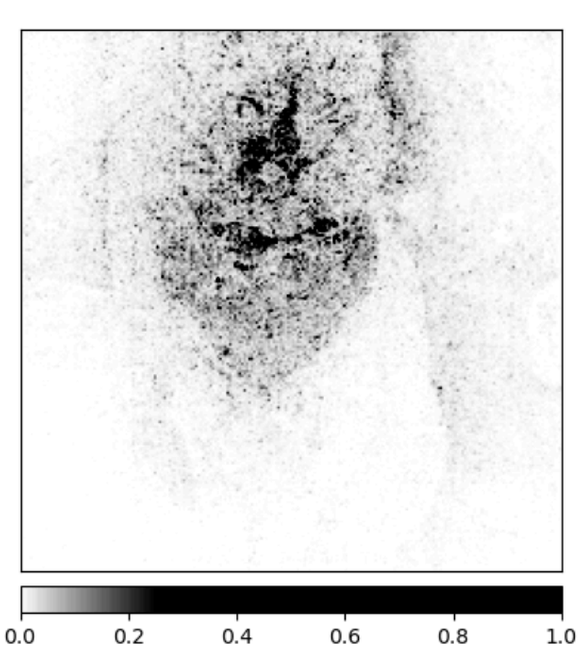
Interpretability/Explainability: Current State of Affairs

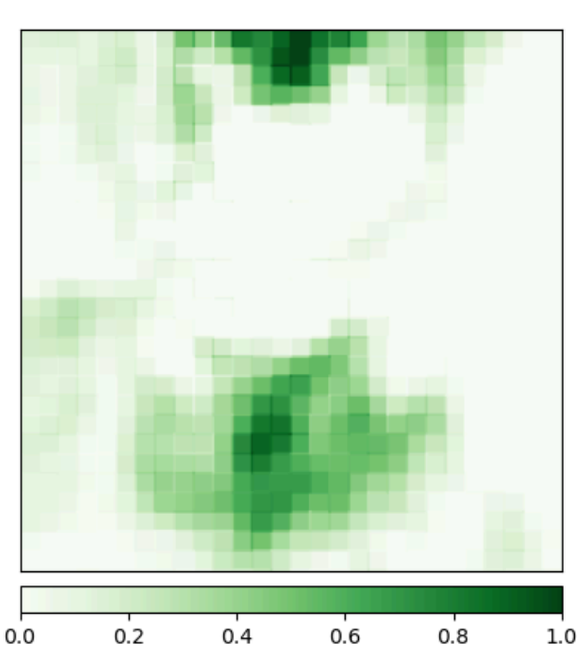
- There are lots of "explanation" approaches that can be used after learning a deep net to try to understand what has been learned
  - Many of these are implemented in the Python package Captum developed by Meta/Facebook: https://captum.ai/

ResNet-18 (a CNN) predicts my cat to be an "Egyptian cat"

What pixels are important for prediction?







Crop image
(many CNNs need the input image to be a specific size)

These are the answers from 3 different explanation models (they give different answers!)

Warning: there's a **lot** of debate as to how much we should actually trust these explanations, as they can often be misleading

Interpretability/Explainability: Current State of Affairs

There are neural net architectures that *by design are interpretable* (e.g., prototypical part networks, neural topic models, neural decision tree models…)

- No separate explanation approach needed since model directly provides explanation
- My opinion: if you really care about interpretability/explainability, then you're better off using this sort of model

Unfortunately, deep nets with state-of-the-art prediction accuracy tend to be difficult to interpret

It's important to distinguish between tasks where interpretability is important vs ones where it's not as important

Unstructured Data Analysis

Question → Data → Finding Structure → Insights

The dead body

Th

by a domain expert

The evidence

have to collect more evidence!

Puzzle solving,

analysis

When? Where?

?

r

catchable?

Answer original question

Becoming good at data scientist requires you to think like a detective!

Much like how some murder mysteries are hard to solve, many data analysis problems (unstructured or not) are hard to solve too!

There isn't always a follow-up prediction problem to solve!

UDA involves *lots* of data → write computer programs to assist analysis

Some Parting Thoughts

- Remember to visualize steps of your data analysis pipeline
  - Helpful in debugging & interpreting intermediate/final outputs
- Very often there are *tons* of models/design choices to try
  - Come up with **quantitative metrics** that make sense for your problem, and use these metrics to evaluate models (think about how we chose hyperparameters!)
    - But don't blindly rely on metrics without **interpreting results in the context of your original problem!**
- Often times you won't have labels! If you really want labels:
  - Manually obtain labels (either you do it or crowdsource)
  - Set up "self-supervised" learning task
- There is a *lot* we did not cover — keep learning!

Want to Learn More?

- Some courses at CMU:
  - Natural language processing (analyze text): 11-611
  - Computer vision (analyze images): 16-720
  - Deep learning: 11-785, 10-707
  - Deep reinforcement learning: 10-703
  - Math for machine learning: 10-606, 10-607
  - Intro to machine learning at different levels of math: 10-601, 10-701, 10-715
  - Machine learning with large datasets: 10-605

- One of the best ways to learn material is to teach it!

  Apply to be a TA for me next term!