Some history

Last time neural nets were popular:

in 80s and 90s, people worked out the backpropagation algorithm

↓↓

allowed the training of neural nets with hidden layers

↳ but pretty much all the neural nets with hidden layers that were trained with only 1 hidden layer

↳ why?

people couldn't really get things to work with more hidden layers

↳ that only started to change in the resurgence of deep learning

2006. "Greedy Layer-wise Training of Deep networks" pointed out these problem

[2010's decade, people actually figure out how we could have deep neural networks that actually worked]

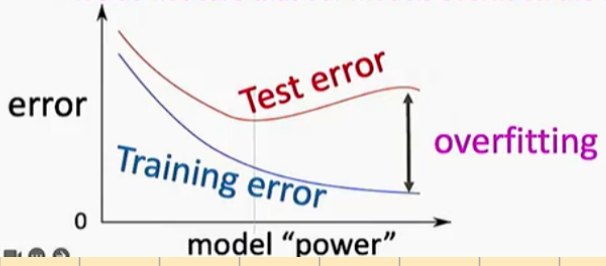
↓↓

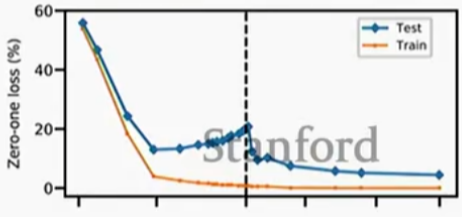
Better Regularization

- A full loss function includes **regularization** over all parameters θ, e.g., L2 regularization:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) + \lambda \sum_k \theta_k^2$$

- Classic view: Regularization works to prevent overfitting when we have a lot of features (or later a very powerful/deep model, etc.)
- Now: Regularization produces models that generalize well when we have a "big" model
  - We do not care that our models overfit on the training data, even though they are **hugely** overfit





Making the parameters numerically small is meant to lessen the extent to which you overfit on your training data (what people previously believed)

Now, people care about the models that will generalize well to different data.

if we train a huge network now on training set, we can essentially train them to get 0 loss, cause we can make it memorize the entire traing set

So, we don't need to constrain the size of weight

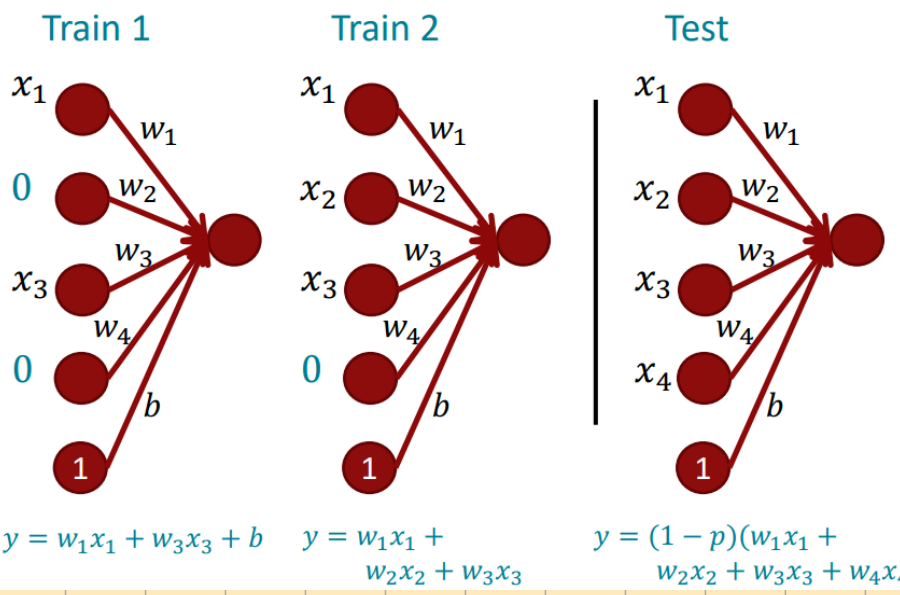
L1 / L2 regularization

⇓

Dropout regularization (favorite now)

Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov 2012/JMLR 2014)

- During training
  - For each data point each time:
    - Randomly set input to 0 with probability p "dropout ratio" (often $p = 0.5$ except $p - 0.15$ for input layer) via dropout mask
- During testing
  - Multiply all weights by $1 - p$
  - No other dropout



technically, it achieved by using a dropout mask consisting of Bernouli random variables

In result, it force the network to not rely on any single neuron or combination of neurons, because any neuron may "strike"(罢工) at any time

Why during testing, it multiply all weights by $1-p$?

since only a $(1-p)$ fraction of neurons are active during training, while 100% of neurons are active during testing. So, to compensate for this discrepency and ensure consistent training and testing scales, we should multiply $(1-p)$

Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov 2012/JMLR 2014)

Why does it work?

Prevents Feature Co-adaptation = Good Regularization! **Use it everywhere!**

Let's talk through an example..

- Training time: at each instance of evaluation (in online SGD-training), randomly set ~50% (p%) of the inputs to each neuron to 0 (less for the first layer)
- Test time: halve the model weights (now twice as many)
- No co-adaptation: A feature cannot only be useful in the presence of particular other features

In a single layer: A kind of middle-ground between Naïve Bayes (all feature weights set independently) and logistic regression models (weights are set in the context of all others)

- Can be thought of as a form of model bagging (i.e., like an ensemble model)
- Nowadays usually thought of as strong, feature-dependent regularizer [Wager, Wang, & Liang 2013]

What's co-adaptation?

The network may have some neurons specialized in "correcting" the errors of other neurons, forming a fragile dependency. Dropout breaks this dependency, forcing each neuron to provide useful information as independly as possible

Why does it work?

1. Prevent co-adaptation
2. Approximately integrating an exponential number of sub-models

   a network with N neuron, can generate 2^N sub-net structure, and these subset learning at same time.

"Vectorization"

- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix:

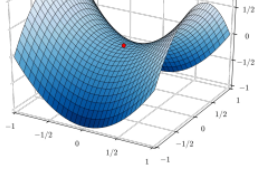
```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- for loop: 1000 loops, best of 3: **639 µs** per loop
  Using single a C x N matrix: 10000 loops, best of 3: **53.8 µs** per loop
- Matrices are awesome!!! Always try to use vectors and matrices rather than for loops!
- The speed gain goes from 1 to 2 orders of magnitude with GPUs!

Parameter Initialization

- You normally must initialize weights to small random values (i.e., not zero matrices!)
  - To avoid symmetries that prevent learning/specialization



- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
- Initialize **all other weights** ~ Uniform(–*r*, *r*), with *r* chosen so numbers get neither too big or too small
  [later, the need for this is removed with use of layer normalization]
- Xavier initialization has variance inversely proportional to fan-in n_{in} (previous layer size) and fan-out n_{out} (next layer size):

$$\mathrm{Var}(W_i) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

Symmetries may make model stuck and stay in the one place
So we almost always want to set all the weights to very small random numbers

Optimizers

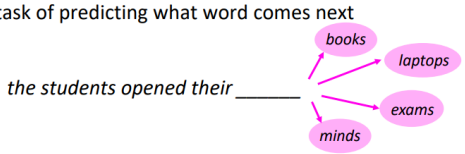
- Usually, plain SGD will work just fine!
  - However, getting good results will often require hand-tuning the learning rate
    - E.g., start it higher and halve it every *k* epochs (passes through full data, **shuffled** or sampled)
- For more complex nets, or to avoid worry, try more sophisticated "adaptive" optimizers that scale the adjustment to individual parameters by an accumulated gradient
  - These models give differential per-parameter learning rates
    - Adagrad ← Simplest member of family, but tends to "stall early"
    - RMSprop
    - Adam ← **A fairly good, safe place to begin in many cases**
    - AdamW
    - NAdamW ← Can be better with word vectors (W) and for speed (Nesterov acceleration)
    - …
  - Start them with an initial learning rate, around 0.001 ← Many have other hyperparameters

Idea of optimizer

For each parameter, they're accumulating a measure of what the gradient has been in the past And they've got some idea of the scale of the gradient, the slope, for a particular prameter. Then they're using that to decide how much you move the learning rate at each time step

2. Language Modeling

- Language Modeling is the task of predicting what word comes next



- More formally: given a sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$, compute the probability distribution of the next word $\boldsymbol{x}^{(t+1)}$:

$$P(\boldsymbol{x}^{(t+1)} | \ \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

where $\boldsymbol{x}^{(t+1)}$ can be any word in the vocabulary $V = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{|V|}\}$

- A system that does this is called a Language Model

Language Modeling

- You can also think of a Language Model as a system that assigns a probability to a piece of text
- For example, if we have some text $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$, then the probability of this text (according to the Language Model) is:

$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) = P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} | \ \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} | \ \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)})$$
$$= \prod_{t=1}^{T} P(\boldsymbol{x}^{(t)} | \ \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})$$

This is what our LM provides

In particular

Language models have been central to NLP at least since the 80s, the idea of them goes back to at least the 50s. They weren't something that got invented in 2022 with ChatGPT

Language model

before neural language model

n-gram Language model (1975 ~ 2012)

n-gram Language Models

the students opened their _____

- **Question**: How to learn a Language Model?
- **Answer** (pre- Deep Learning): learn an *n*-gram Language Model!

- **Definition:** An *n*-gram is a chunk of *n* consecutive words.
  - unigrams: "the", "students", "opened", "their"
  - bigrams: "the students", "students opened", "opened their"
  - trigrams: "the students opened", "students opened their"
  - four-grams: "the students opened their"

- **Idea:** Collect statistics about how frequent different n-grams are and use these to predict next word.

n-gram Language Models

- First we make a Markov assumption: $x^{(t+1)}$ depends only on the preceding *n*-1 words

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}}) \quad \text{(assumption)}$$

$$= \frac{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})} \quad \text{(definition of conditional prob)}$$

prob of a n-gram / prob of a (n-1)-gram

- **Question:** How do we get these *n*-gram and (*n*-1)-gram probabilities?
- **Answer:** By counting them in some large corpus of text!

$$\approx \frac{\text{count}(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{\text{count}(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})} \quad \text{(statistical approximation)}$$

n-gram Language Models: Example

Suppose we are learning a 4-gram Language Model.

~~*as the proctor started the clock, the*~~ *students opened their* _____

discard — condition on this

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count}(\text{students opened their } \boldsymbol{w})}{\text{count}(\text{students opened their})}$$

For example, suppose that in the corpus:

- "students opened their" occurred 1000 times
- "students opened their books" occurred 400 times
  - → P(books | students opened their) = 0.4
- "students opened their exams" occurred 100 times
  - → P(exams | students opened their) = 0.1

Should we have discarded the "proctor" context?

eg:

2 problems of n-gram LM

Sparsity Problems with n-gram Language Models

Sparsity Problem 1

Problem: What if *"students opened their w"* never occurred in data? Then *w* has probability 0!

(Partial) Solution: Add small δ to the count for every $w \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count}(\text{students opened their } \boldsymbol{w})}{\text{count}(\text{students opened their})}$$

Sparsity Problem 2

Problem: What if *"students opened their"* never occurred in data? Then we can't calculate probability for *any w*!

(Partial) Solution: Just condition on *"opened their"* instead. This is called *backoff*.

Note: Increasing *n* makes sparsity problems *worse*. Typically, we can't have *n* bigger than 5.

Storage Problems with n-gram Language Models

Storage: Need to store count for all *n*-grams you saw in the corpus.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count}(\text{students opened their } \boldsymbol{w})}{\text{count}(\text{students opened their})}$$

Increasing *n* or increasing corpus increases model size!

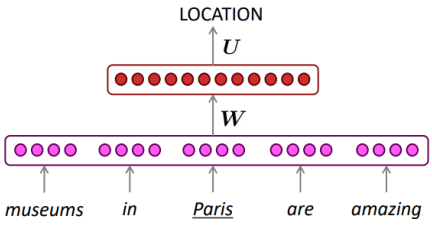
Markov assumption's main idea

The state of an event is only related to the states of a finite number of events before it (usually 1 or 2), and has nothing to do with further history

Neural language model

How to build a *neural* language model?

- Recall the Language Modeling task:
  - Input: sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$
  - Output: prob. dist. of the next word $P(x^{(t+1)} | x^{(t)}, \ldots, x^{(1)})$
- How about a window-based neural model?
  - We saw this applied to Named Entity Recognition in Lecture 2:



27

At first, people developed "fixed-window neural Language model"

~~as the proctor started the clock~~ the students opened their _____

discard

fixed window

A fixed-window neural Language Model

output distribution

$\hat{y} = \text{softmax}(Uh + b_2) \in \mathbb{R}^{|V|}$

hidden layer

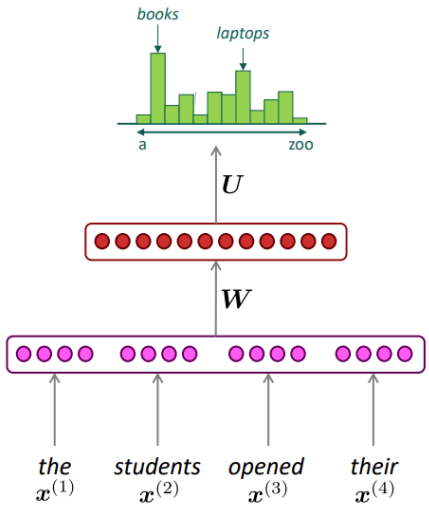
$h = f(We + b_1)$

concatenated word embeddings

$e = [e^{(1)}; e^{(2)}; e^{(3)}; e^{(4)}]$

words / one-hot vectors

$x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}$

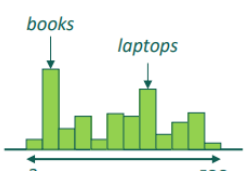


A fixed-window neural Language Model

Approximately: Y. Bengio, et al. (2000/2003): A Neural Probabilistic Language Model

Improvements over *n*-gram LM:

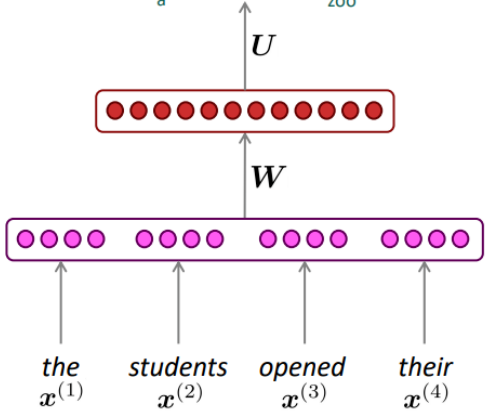
- No sparsity problem
- Don't need to store all observed *n*-grams



Remaining **problems**:

- Fixed window is too small
- Enlarging window enlarges W
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W. No symmetry in how the inputs are processed.

We need a neural architecture that can process *any length input*



30

eg: if we change "students"'s position to $x^{(1)}$/$x^{(3)}$/$x^{(4)}$, it will be multiplied by completely different weights in W, though the target position is still "students"

What we want to achieve ?

a different kind of neural architecture that can process any length of input and can use the same parameters to deal with same word wherever it occurs.

Recurrent neural networks (RNN)

in some sense, word2vec is the first neural network architecture we saw, and feedforward or full connected layer classic neural networks is the second

The core idea:

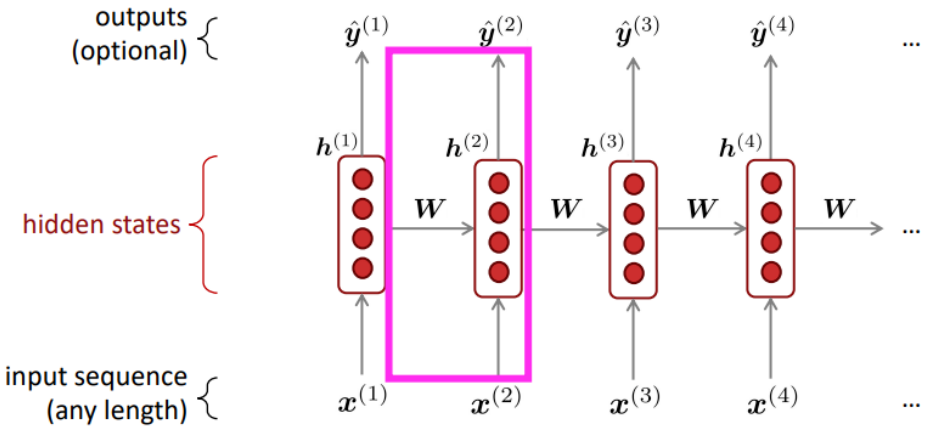
It uses a recurrent connection that allows the network to pass information from previous time steps to current one (you've got a set of weight that are going to be applied through successive moments in time, i.e. sets of position

in the text, so you do that, you're going to update the parameters)
This enables it to process sequential data like sentences or time series, where the output depends not only on the current input but also on the historical context.

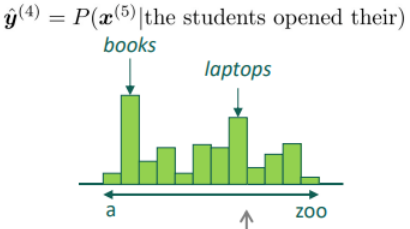
3. Recurrent Neural Networks (RNN)

A family of neural architectures

Core idea: Apply the same weights *W repeatedly*



A Simple RNN Language Model



output distribution

$$\hat{y}^{(t)} = \text{softmax}\left(Uh^{(t)} + b_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right)$$

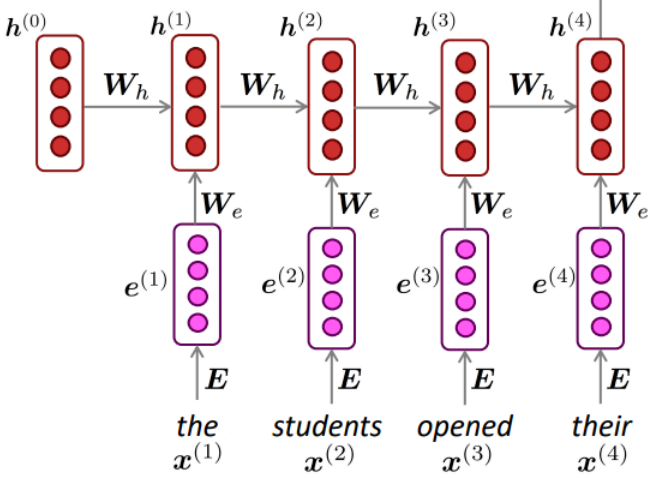
$h^{(0)}$ is the initial hidden state

word embeddings

$$e^{(t)} = Ex^{(t)}$$

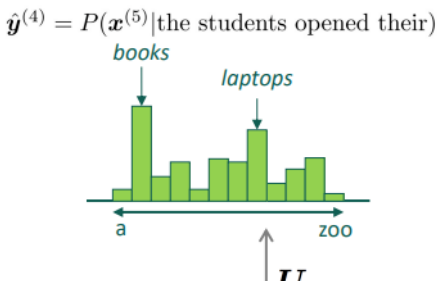
words / one-hot vectors

$$x^{(t)} \in \mathbb{R}^{|V|}$$



Note: this input sequence could be much longer now!

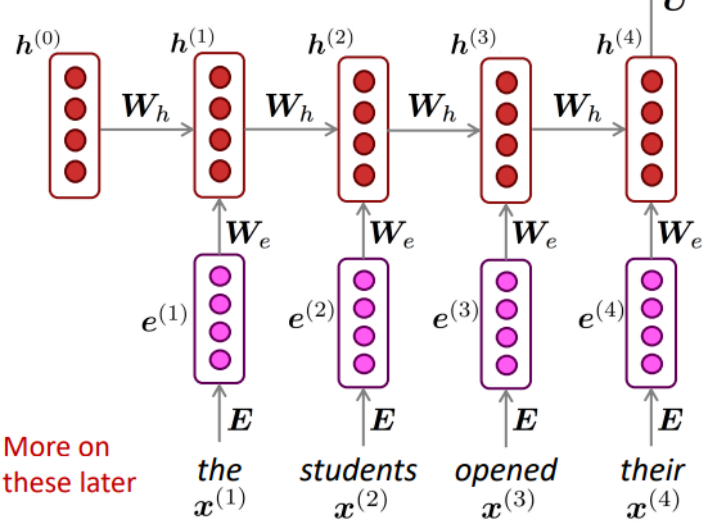
RNN Language Models



RNN **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from

many steps back
- Model size doesn't increase for longer input context
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.



RNN **Disadvantages**:
- Recurrent computation is slow
- In practice, difficult to access information from many steps back

More on these later

it just like we still use for-loop to compute
↳ it led RNN fall out of favor

It means your memory of it ges more and more distant. And it's the words that you saw recently that dominate the hidden state, In some sense, it's right, because human also do the same things. However, RNN will forget stuff from further back too quickly
gradient vanish

Train the RNN

Training an RNN Language Model

- Get a big corpus of text which is a sequence of words $x^{(1)}, \ldots, x^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\boldsymbol{y}}^{(t)}$ for *every step t*.
  - i.e., predict probability dist of *every word*, given words so far

- Loss function on step *t* is cross-entropy between predicted probability distribution $\hat{\boldsymbol{y}}^{(t)}$, and the true next word $\boldsymbol{y}^{(t)}$ (one-hot for $x^{(t+1)}$):

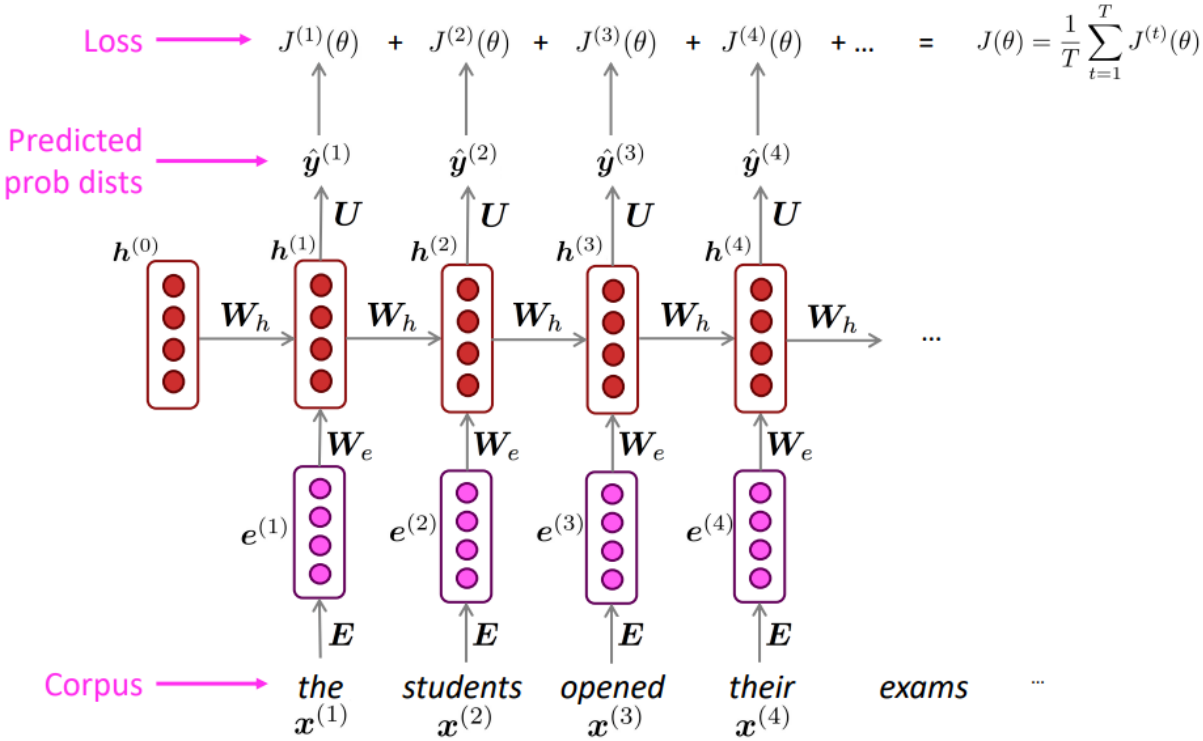
$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

- Average this to get overall loss for entire training set:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

Training an RNN Language Model

"Teacher forcing"



Teacher forcing

It's a training technique used for RNNs, particularly in sequence-to-sequence models. During training, instead of using the model's own prediction from the previous time step as the current input, it forces the use of the ground truth data from the training set as the input for the next step

This method accelerates model convergence and improves traing stability

Forward propagation of RNN

n : the dimension of hidden layer

d : the dimension of one-hot vector (the length of corpus)

U

W_e : input weight $W_e \in R^{n \times d}$

W_h : recurrent weight $W_h \in R^{n \times n}$

U : output weight $U \in R^{d \times n}$

input / output vector $\in R^d$

hidden layer $\in R^n$

bias $\begin{cases} b_h \in R^d \\ b_y \in R^n \end{cases}$

$y^{(t)}$: predict y_t : grand true

Timestep $t=1$

1. input vector $x^{(1)}$
2. $h^{(1)} = \tanh(We * x^{(1)} + Wh * h_0 + b_h)$; then store $h^{(1)}$
3. $\hat{y}^{(1)} = \text{softmax}(U * h^{(1)} + b_y)$
4. compute loss : $J^{(1)} = -\sum y_1 * \log \hat{y}^{(1)}$, and only one place of y_1 is 1.

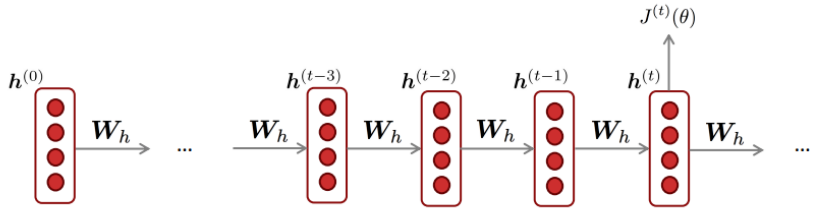
eg. If the true next word is at the place of m (namely $[0, 0, 0, \ldots, 1, 2, \ldots]$, m)

$J^{(1)} = -\log(\hat{y}^{(m)})$

Timestep $t=2$ ----

$t=$ ----

Backpropagation for RNNs



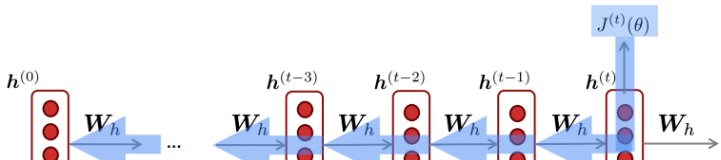
Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix W_h ?

Answer: $$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)}$$

"The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears"

Why?

Training the parameters of RNNs: Backpropagation for RNNs



In practice, often "truncated" after ~20 timesteps for training efficiency reasons

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)}$$

equals equals equals equals equals

W_h

Question: How do we calculate this?

Answer: Backpropagate over timesteps $i = t, \ldots, 0$, summing gradients as you go. This algorithm is called **"backpropagation through time"** [Werbos, P.G., 1988, *Neural Networks* **1**, and others]

Apply the multivariable chain rule:

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)} \frac{\partial W_h|_{(i)}}{\partial W_h}$$ (= 1)

$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)}$$

$z_{y_t} = U * h^{(t)} + by$

$z_{ht} = We * x^{(t)} + Wh * h^{(t-1)} + bh$

What should we compute?

$$\frac{\partial J}{\partial We}, \frac{\partial J}{\partial Wh}, \frac{\partial J}{\partial U}, \frac{\partial J}{\partial bh}, \frac{\partial J}{\partial by}$$

how?

step t=1. the crucial step, to compute $\frac{\partial J}{\partial z_{yt}}$

why compute $\frac{\partial J}{\partial z_{yt}}$ instead of computing $\frac{\partial J}{\partial y^t}$ first

↳ because this is a Softmax + Crossentropy loss

the gradient is very easy for z_{yt}

$$\frac{\partial J}{\partial z_{yt}} = \hat{y}_t - y_t = \delta_{y_t} \quad \text{(the upstream gradient)}$$

2. compute $\frac{\partial J}{\partial U}$ and $\frac{\partial J}{\partial by}$

$$\frac{\partial J}{\partial U} = \delta_{y_t} \cdot h^{(t)}$$

$$\frac{\partial J}{\partial by} = \delta_{y_2}$$

3. $\frac{\partial J}{\partial h_t} = U^T \cdot \delta_t$

4. $\frac{\partial J}{\partial z_{ht}} = \frac{\partial J}{\partial t} \circ (1 - h_t^2) = \delta_{ht}$ $\left(\frac{d \tanh}{dx} = 1 - \tanh^2(x)\right)$

5. $\frac{\partial J}{\partial We} = \delta h_t \cdot x^{(t)T}$

$\frac{\partial J}{\partial Wh} = \delta h_t \cdot h_{t-1}^T$

$\frac{\partial J}{\partial bh} = \delta h_t$

step t-1:

1, 2

$J_1 : h_{t-1} \to z_{y_{t-1}} \to J_{t-1}$

$J_2 : h_{t-1} \to z_{h_t} \to h_t \to \cdots J_t$

#3: $\frac{\partial J}{\partial h_{t-1}} = \underbrace{(U^T \cdot \delta_{y_{t-1}})}_{\text{from } J_{t-1}} + \underbrace{(W_h^T \cdot \delta_{h_t})}_{\text{from } J_t}$

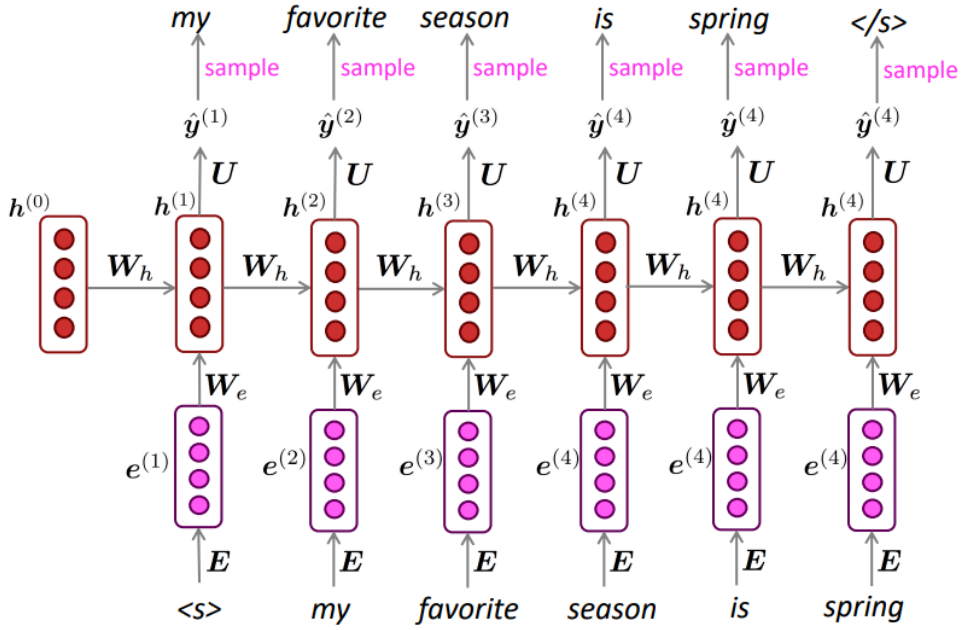
4. $\nabla W_e = \frac{\partial J}{\partial W_e^{(t-1)}} + \frac{\partial J}{\partial W_e^{(t)}}$

$\nabla W_h = \frac{\partial J}{\partial W_h^{(t-1)}} + \frac{\partial J}{\partial W_h^{(t)}}$

$\nabla b_h = \frac{\partial J}{\partial b_h^{(t-1)}} + \frac{\partial J}{\partial b_h^{(t)}}$

Generating with an RNN Language Model ("Generating roll outs")

Just like an n-gram Language Model, you can use a RNN Language Model to generate text by repeated sampling. Sampled output becomes next step's input.



44

hidden state

the "memory" of a RNN, it summarizes all the the relevant information the network has seen so far in the sequence.

Initializing hidden state

Initialize the vector h_0 (default h_0 = zero vector)