ECE 1786 Lecture #2

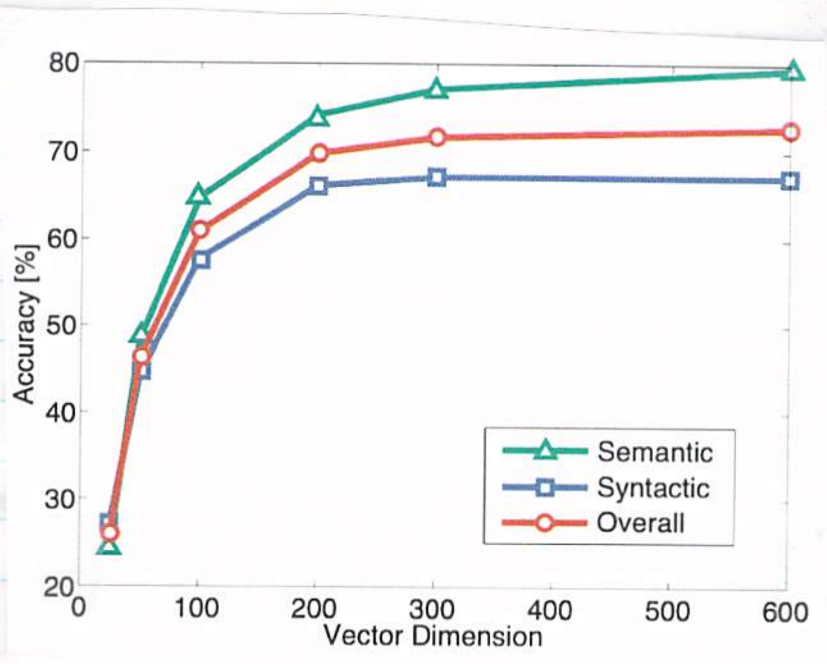
Last Day: Word Embeddings Properties & Meaning Extraction.

Work-in-Flight:
- Assignment 1 due next week.
- Fill out survey if ~~needed~~ have not — new registrants

Today: How Word Embeddings are Created.

Recall:
- Word embeddings represent meaning of words in numbers.
- helps neural networks deal with ambiguity in language
- ~~Some~~ embeddings that are close mean that the associated words are close in meaning
- also other associations Queen − King = Woman − Man ++
- → You should have done Assign 1 parts 1 & 2 now.

- last week we discussed (a little) how big dimension size should be; diminishing returns beyond 300 according to Glove paper. by Pennington et. al.



How to create (train) vectors?
- a clever + complex idea that begins with Bengio → Mikolov.

Big picture of method: We train a neural network to make a prediction based on the meaning of a word. Inside the neural network that meaning will be encoded. The training of the network will cause the encoding to be learned.

That encoding ~~ ~~ IS the embedding/vector.

(apologies for now using 3 terms that are the same thing: encoding/embedding/vector)
→ relates to auto-encoders

So, what is that prediction task, and where does the data come from? Where do the labels come from?

* The Prediction Task: Does word A "belong" with word B?
i.e.. are they related somehow?
Let's contemplate this by → are they often used together
Considering these three sentences:

1. The mathematician ran to the store.
2. The engineer ran to the store.
3. The mathematician solved the problem.

Sentences 1 & 2 imply similarity ~~of~~ between engineer & mathematician. which is what?
⇒ not surprising that this is a sensible sentence
4. "The engineer solved the problem"

- this is an example of the "Distributional Hypothesis":

"Words appearing in similar contexts are related"

- since we want to predict which words are related, we have a ready made dataset of examples and labels: every sentence ever written!
  - an enormous dataset! yippee!!

- let's consider some simple sentences taken from the SimpleCorpus.txt in Assignment ~~1~~ 3 in this course

- looks like this: I hold a dog. She holds a dog. He holds a dog. I rub the dog. She rubs the cat....

- these even simple sentences give some clues to the meaning of the words used, and which are related → what are they?

- we can generate NN training examples of words that are related by selecting words that are 'near' each other in 'correct' (valid) written sentences.
  - near ≜ within a few (1-3) words (on either side) of target word

- for example, consider "I hold a dog"; if we take 'near' to be words ±2 words away from target then the training examples of pairs of related words are

(I, hold)	(hold, I)	(a, I)	(dog, hold)
(I, a)	(hold, a)	(a, hold)	(dog, a)
	(hold, dog)	(a, dog)	
↑			
I	hold	a	dog

Target word + context word.

- clearly can make a lot of those.

- want to build a NN predictor that given a word pair (wordA, wordB) can predict wordB given wordA:
  "target word" "context word"

  input: word A → (NN.) → output: word B.

  - for now, assume that words can only come from a specific, limited vocabulary → define.

- a key part of this is how the input is represented; what is the simplest way to represent all the words in a vocabulary of size V? – 1 hot |V|
  – discuss 1-hot
- we know that we want to represent the words by a vector of size << vocabulary size |V| eg
  (that's what 1st lecture was about)

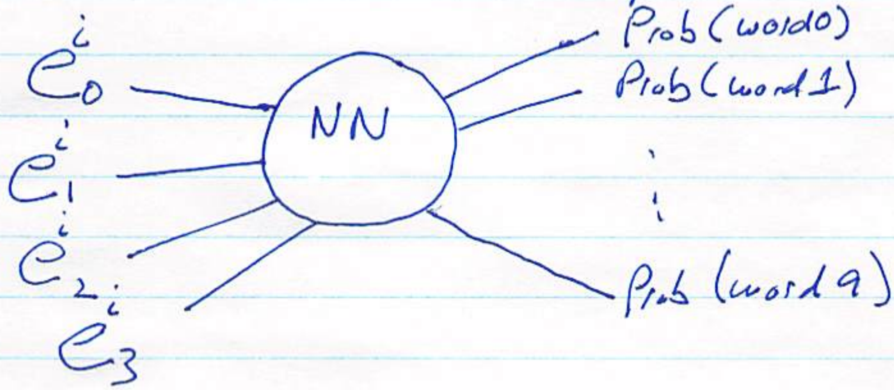
- let's consider a simple example similar to Assignment 1, part 3.
  - let's say the vocabulary size |V| = 10 — 10 words! typical vocab ≥ 2K–10K.
  - let's assume the embedding size has dim = 4 (typically 1st 50+1)
    (vector, encoding)
  - so that implies we've ~~got~~ want 10 embeddings of size 4 each;
  - these are typically stored in a matrix like so

$$
\begin{array}{ccccc}
\text{words } 0 & 1 & 2 & \cdots & 9 \\
\left[\begin{matrix} e_0^0 \\ e_1^0 \\ e_2^0 \\ e_3^0 \end{matrix}\right. & \begin{matrix} e_0^1 \\ e_1^1 \\ e_2^1 \\ e_3^1 \end{matrix} & \begin{matrix} e_0^2 \\ e_1^2 \\ e_2^2 \\ e_3^2 \end{matrix} & \begin{matrix} \cdots \\ \cdots \\ \cdots \\ \cdots \end{matrix} & \left.\begin{matrix} e_0^9 \\ e_1^9 \\ e_2^9 \\ e_3^9 \end{matrix}\right] \\
\text{"she"} & \text{"he"} & \text{"hold"} & & \text{"dog"}
\end{array}
$$

→ dim × |V| "embedding matrix".

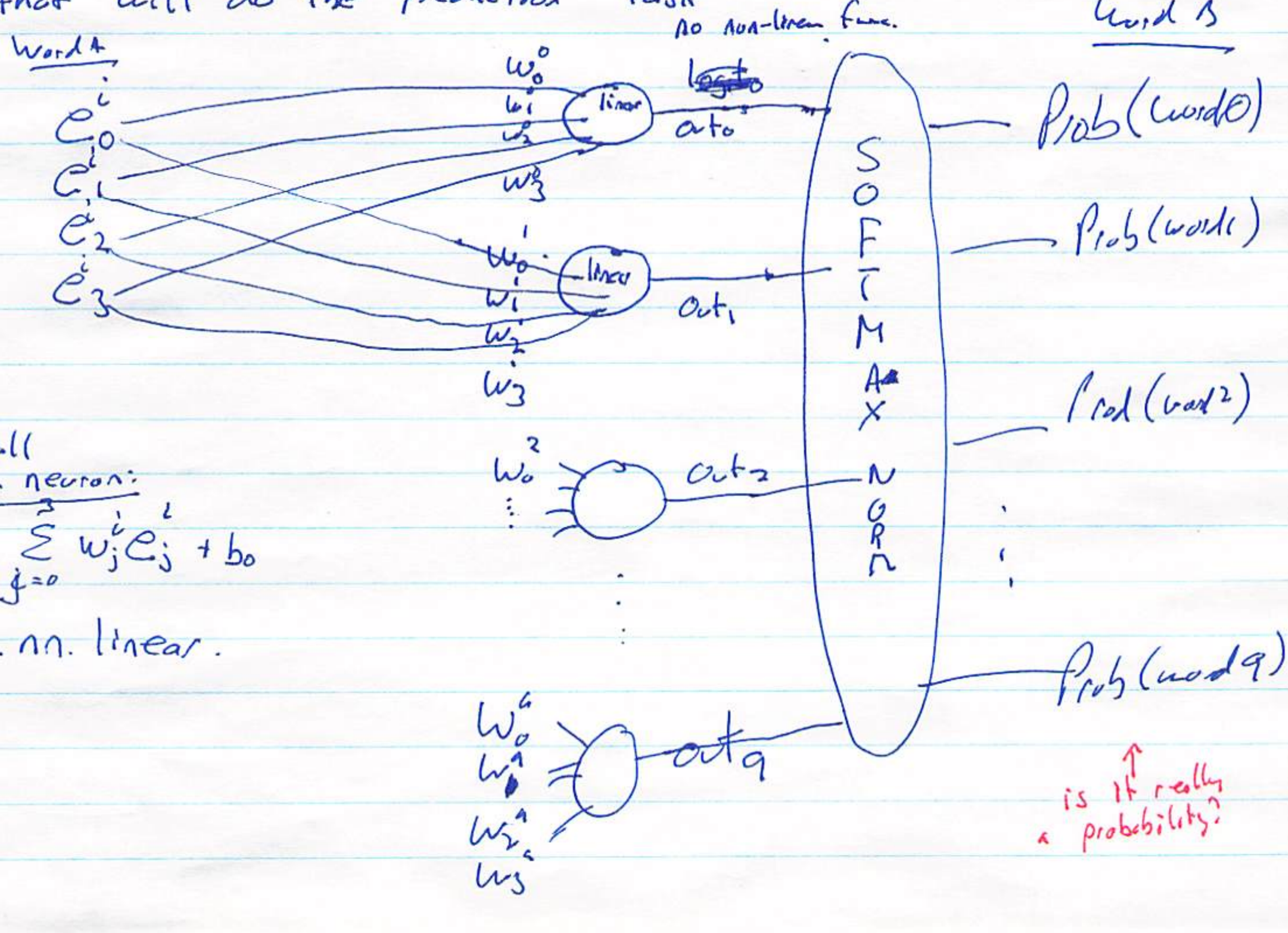
- Key: these values will be randomly initialized and learned through training just like weights & biases
  (can pose this so that these are exactly weights, but didn't)

- so the input to the NN will be these 4 values ($e_0 \to e_3$) associated with word A in the embedding matrix
  - but what should the output be?



- where i is the word number of word A (aka 'target' word)

here is the
- ~~lets draw what~~ the $|V|$-neuron neural network ~~looks like~~: that will do the prediction task



recall linear neuron:

$$out_0 = \sum_{j=0}^{3} w_j^i e_j^i + b_0$$

$\to$ torch.nn.linear.

- So to train this network, you would present many examples of (wordA, wordB) pairs.
  - → wordA is the e_j^i target
  - → wordB, the context label, is 1-hot encoded in a vector of size $|V| = 10$. → discuss.
  - → ask
  - → Use "cross entropy" loss to train (the -log of correct answer)

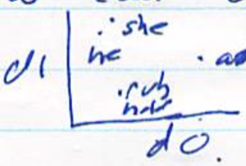
$\boxed{\text{note PyTorch combines softmax + cross-entropy into 1 function}}$ for numerical stability

- Crucial, to repeat: the values of e_j^i are also learned through gradient descent.

- Nuance / improvement: observe the w_j^i
  - these are different word embeddings for each word.
  - ⇒ we don't really need 2 embeddings, so can save computation (½ # parameters) by just using 1
  - ⇒ i.e. can replace w_j^i with e_j^i (the same parameter appears in 2 places in the network)
  - → ~~do this in assignment 1, part 3.~~
  - → a different way to think about this:
  - → to perform the prediction task, "do these two words relate?", just compute the dot product of their word vectors
    - → the higher it is, the more similar they are
    - → just like a convolution kernel.

Assignment 1, Section 3 asks you to train a vocabulary of size 11 on a small corpus with an embedding dimension of 2 — I, he, she, can, rub, hold, dog, cat, and, the, a

- & see if similar words end up 'near' each other on 2-D plot

→ 

→ maybe show this.

- some details: first lemmatize — holds → hold, rubs → rub, rub → rub

words are reduced to their roots.

→ Next: ↑ is too slow, for realistic sizes; want you to experience larger vocab, larger dim, + larger corpus

So a faster way ↳ is called the Skip-Gram method of training word vectors

↳ this section allows you to get back up to speed on training; it illustrates the word embedding concept from "scratch" with very small dimension & small vocabulary & small corpus

AI Pr.

Skip-Gram with Negative Sampling

- rather than predict which of the |V| words in the vocabulary are associated (related) to the target word, (a multi-class classification) — which is large >5K → 300K

- instead, we make a binary prediction as to whether a given pair of words (targetword, context word) are related or not

→ to do so, we need positive examples of related words - exactly the as SKIP-GRAM, above, but also need negative samples of word pairs that are not related. (to properly train a binary classifier)

- so, to train, create the positive examples as above. (with a given window size, etc)

→ to create the negative samples, for each target word, we randomly sample words from the entire corpus — omitting the word itself. - why is this OK? [i.e won't some positive examples be included?]

→ this sampling gives the method its name: skip-gram with negative sampling.

- described in Jurafsky text, Section 6.8.

- How many negative samples:
  - often 2× as many negative than positive.

- the full skip gram method suggests biasing the random sampling of negative examples to avoid the "high frequency" (most common) words in the corpus → left as a bonus in Section 4, Assignment 1 to explore.

- So, the binary prediction, "are these two words related or not" can be expressed as follows:

given target word's vector ~~to~~ $T = (t_0, t_1, \dots t_{d-1})$
context " " $C = (c_0, c_1, \dots c_{d-1})$
where d is the embedding size

then compute dot product $C \cdot T$ → go to 2-10
$= \sum_{i=0}^{d-1} (t_i \cdot c_i) = P$.

→ convert to a probability using sigmoid func

$P(\text{related}) = \sigma(P)$
& then use binary cross entropy to compute loss

i.e. if label $= 1$ (related) loss $= -\log(P)$
$= 0$ (not related) loss $= -\log(1-P)$

why is it log?

- note again this "similarity (related" computation is a dot product C • T.
  - where else have you see dot product operate as a similarity function? (CNN kernel)
- discuss the math of it.
- words that are related. have a higher +ve dot product
- (refer to 2-9)

→ this is an essential observation that is used in the key model we will cover in this course → the Transformer. The Attention modules inside the transformer are based on this relation
  - some people now think of a Transformer as an extension of ~~@~~ a CNN because of the link between kernel convolution and attention.

- Assignment 1, Section 4 asks you to train word vectors for a much larger vocabulary than Section 3 (>2000 words) a much larger dimension (8) and a much larger training corpus (>60K words)
  - used just SG would be too slow.
  -
- uses different tokenization.

- to visualize the results in two dimensions, must "reduce the dimensionality" from 8 to 2 using Principle Component Analysis as described in Section 4 / starter code.