Neural approaches to distributional modeling: word embeddings and word token embeddings

Neural networks for computing distributional vectors

Why use neural networks?

- Increased speed and ease of computing word vectors / word embeddings
  - Are they also better at approximating word meaning and concepts?

    Not so clear, see Lenci et al, A comparative evaluation and analysis of three generations of Distributional Semantic Models, https://link.springer.com/article/10.1007/s10579-021-09575-z
- Vector / embedding for a word in a particular sentence context: study word meaning in context

Getting into the underlying math of the models

This is extremely important! If we want to use neural networks as tools for linguistics, we need to understand the tools.
We need to understand what they are learning, and by what methods, so we can judge what these models can tell us about patterns and regularities in word use.

I can only do a quick sketch of neural models here. If you can, learn more.

By the way…

Don’t use ChatGPT for science.

We don’t know what it has been trained on.

It is continuously training, so results are not exactly replicable.

It may train on the test data you give it, so that next time it will be perfect because it has memorized the data.

It has a mixed form of training that makes it something other than a compressed record of (regularities in) utterances , so it is harder to draw sound conclusions from its performance.

Prediction-based word embeddings

Starting point: Logistic regression

Katrin Erk

Statistical modeling technique

$y \sim x1 + x2 + x3$

- y: dependent variable, yes/no question
- x1, x2, x3: independent variables, predictors

Starting point: Logistic regression

Katrin Erk

Example: The dative alternation.

“Mary gave John the book” (Recipient is NP) vs “Mary gave the book to John” (Recipient is PP):
when do people choose NP vs PP realization?

- “Mary sent the book to the office” — “Mary sent the office the book”:
  when the Recipient is non-animate, PP seems better
- “Mary gave the book that is about an immortal alchemist and his wife living in San Francisco as booksellers to John”:
  when the Theme is long, NP seems better
- Similarly, length of Recipient: “the book to John, this friend of hers whom she hadn’t seen in a long time”
  for long Recipients, PP seems better

RealizationOfRecipient ~ AnimacyOfRecipient + LengthOfTheme + LengthOfRecipient

Starting point: Logistic regression

Katrin Erk

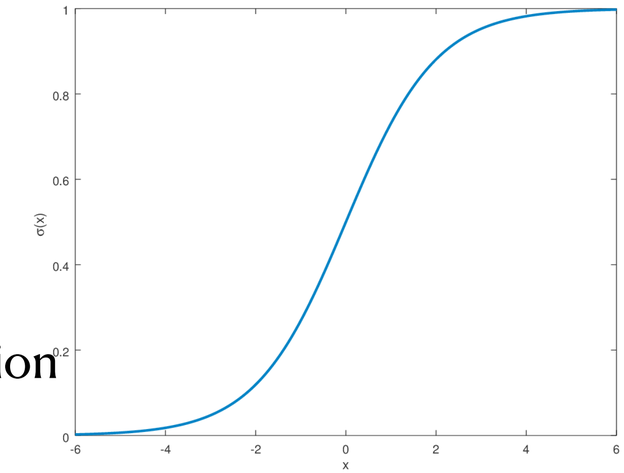
RealizationOfRecipient ~ AnimacyOfRecipient + LengthOfTheme + LengthOfRecipient

What is actually computed is:

P(RealizationOfRecipient = 1) =
sigmoid(w0 + w1 * AnimacyOfRecipient + w2 * LengthOfTheme + w3 * LengthOfRecipient)

- w0, w1, w2, w3: coefficients, weights, fitted to best match data
- whole formula: first linear combination of weighted predictors,
  then transform from a line to an s-shape,
  to get a number between 0 and 1

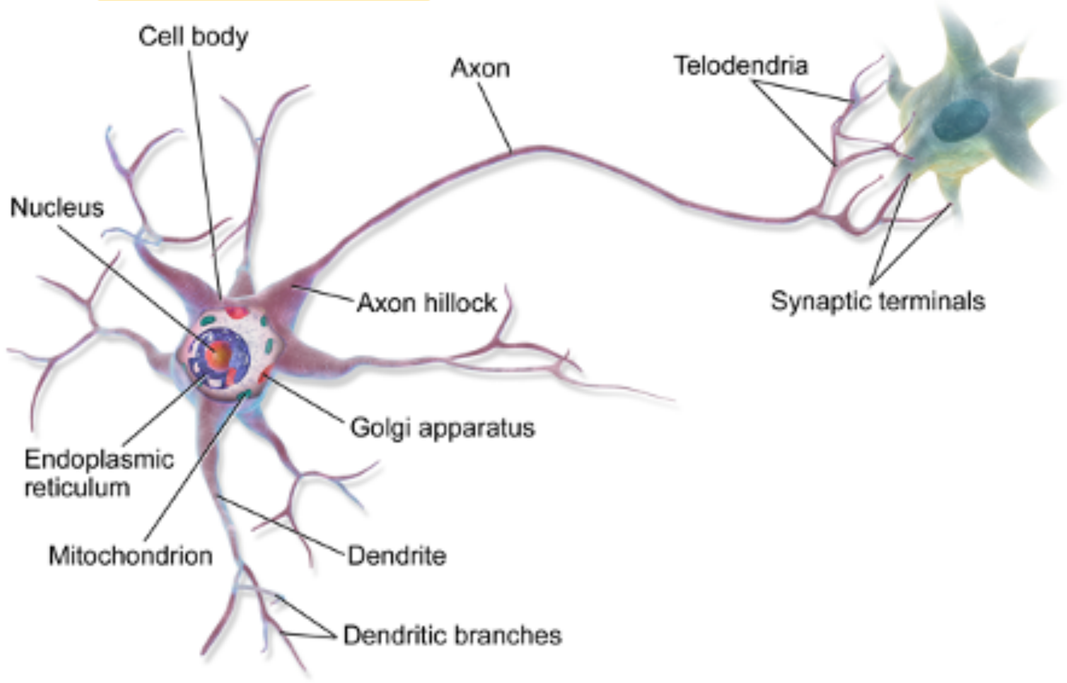
sigmoid function



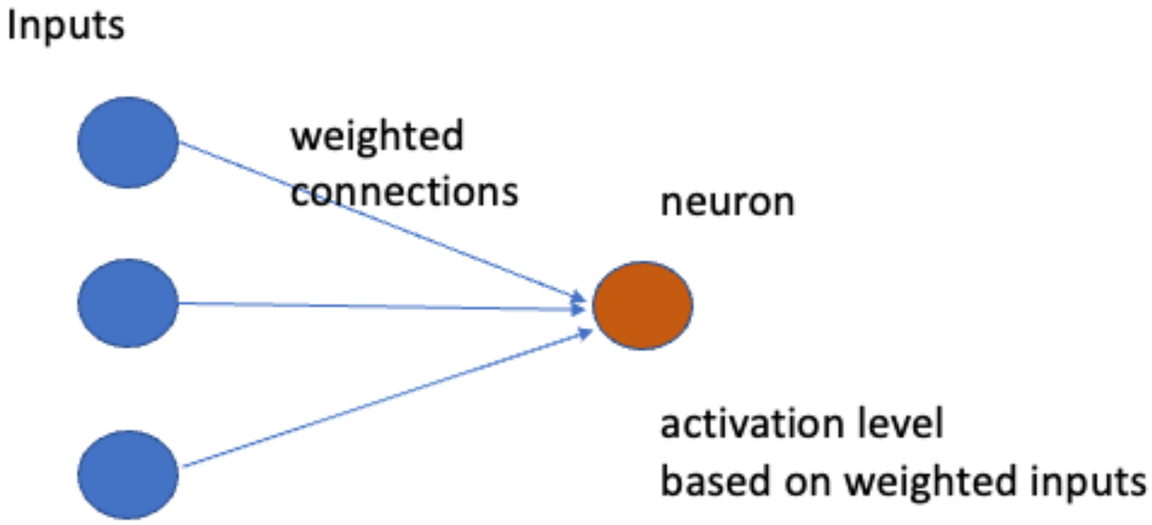
This is an artificial neuron!

McCullough and Pitts, 1940s: Mimicking a human neuron

human neuron

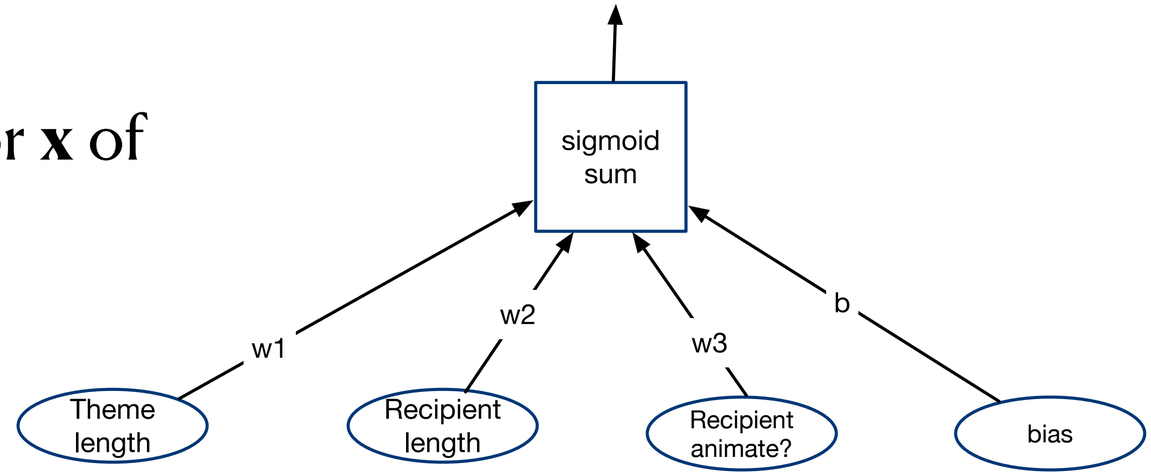


artificial neuron



Generalizing: logistic regression as a neural network

- Using notation of vector **w** of weights, vector **x** of predictors: P(yes) = sigmoid(**wx**)
- Fitting: For every data point (**x**, y), tweak the weights to make the predicted output a bit more like the true y



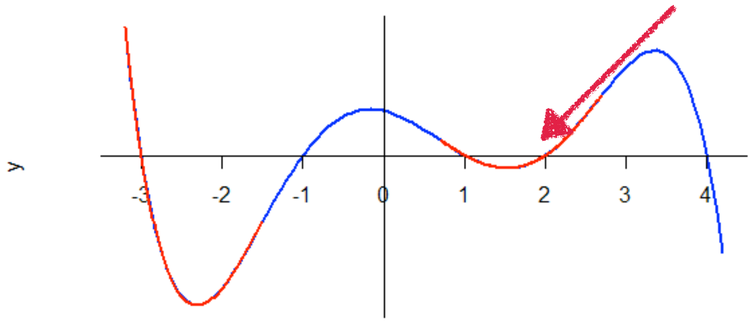
Generalizing over logistic regression

- Logistic regression model as a function:
  - **P(yes)** = **sigmoid(wx)**
- This is:
  - function of inputs x
  - weighted linear combinations of values: wx. weights are “parameters”
  - non-linear transformations: sigmoid
  - Learning: adapt parameters to best match the model’s prediction $\hat{y}$ to the true label y
    - Loss Function L($\hat{y}$, y): How much does prediction $\hat{y}$ diverge from true y? We want to minimize the loss
    - Cross-entropy loss: if actual y = 1, maximize $\hat{y}$, if actual y = 0, maximize $(1 - \hat{y})$,
      or minimize

$$-\log\left(\hat{y}^{y}\ (1-\hat{y})^{1-y}\right)$$

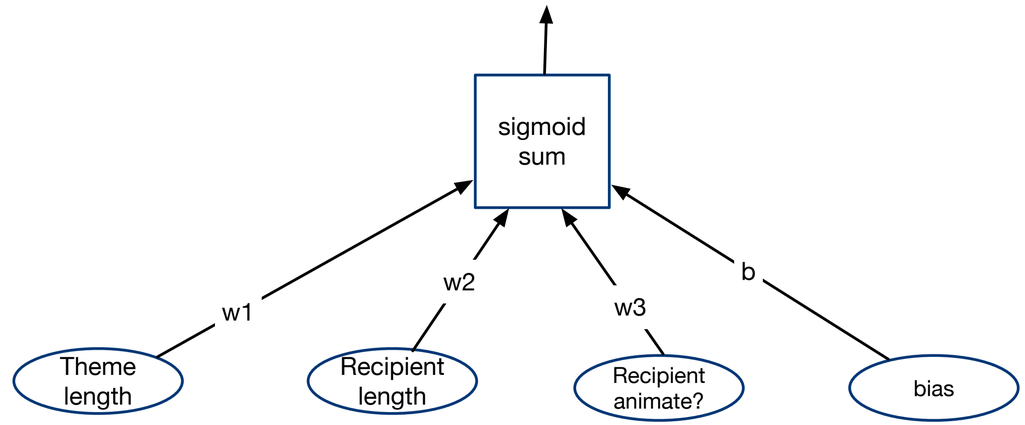
Generalizing over logistic regression

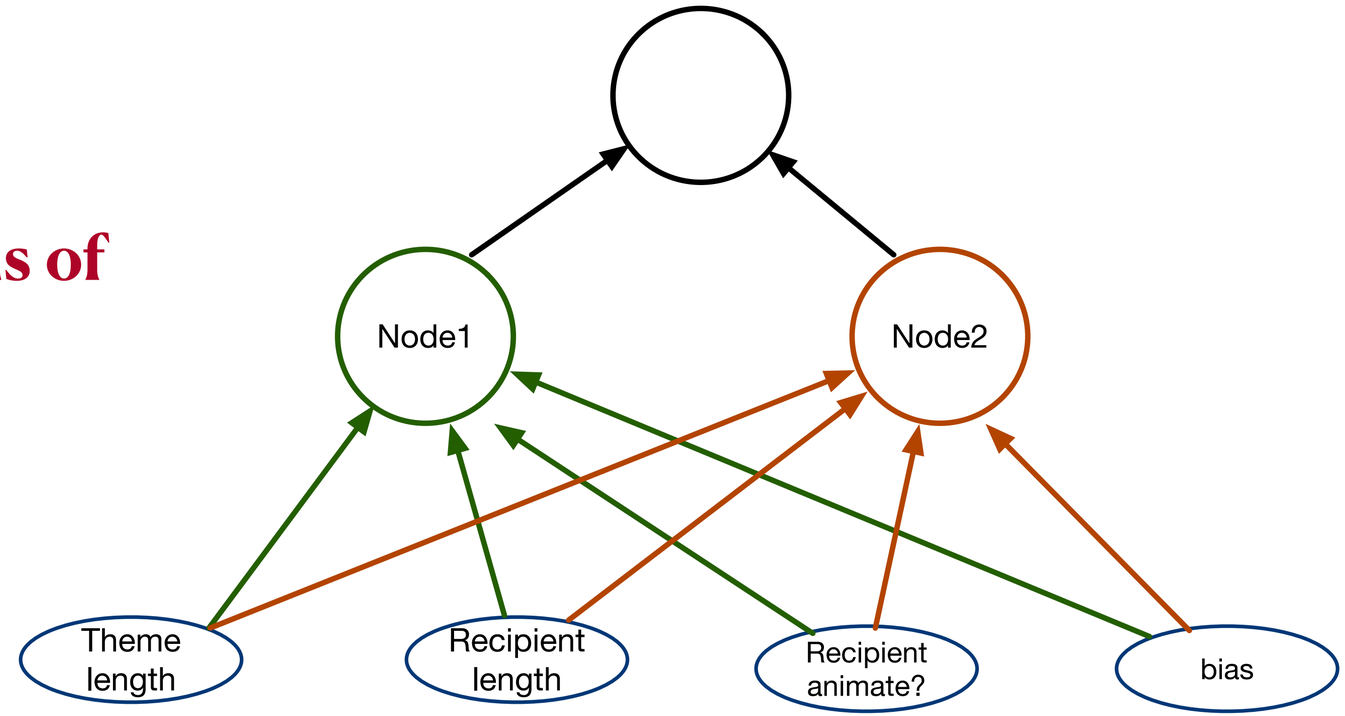
- Loss Function L($\hat{y}$, y): How much does prediction $\hat{y}$ diverge from true y? We want to minimize the loss
  - Cross-entropy loss: if actual y = 1, maximize $\hat{y}$, if actual y = 0, maximize $(1 - \hat{y})$, or minimize $-\log\left(\hat{y}^{y}\ (1 - \hat{y})^{1-y)}\right)$
- Visualize the loss function: in some places it's high, in some places it's low, like a mountain range
- I'm here, what is the quickest way down? Derivative



Building up larger networks from smaller ones

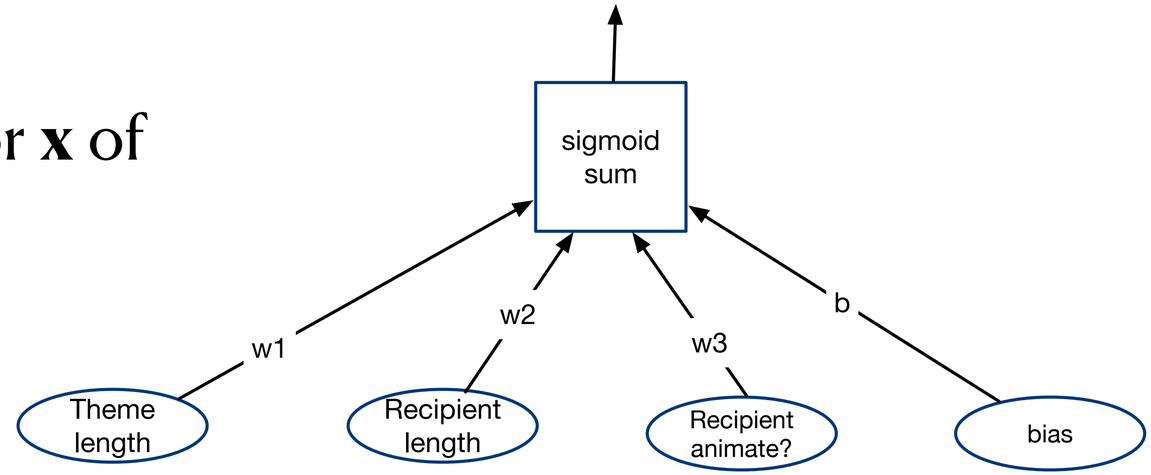
- Green: a logistic regression model
- Red: another logistic regression model
- Larger model learns to combine predictors into internal features
- New **hidden layer**
- **Network learns combinations of inputs to match the task: It can learn its own features!**





Generalizing: logistic regression as a neural network

- Using notation of vector **w** of weights, vector **x** of predictors: P(yes) = sigmoid(**wx**)
- Fitting: For every data point (**x**, y), tweak the weights to make the predicted output a bit more like the true y

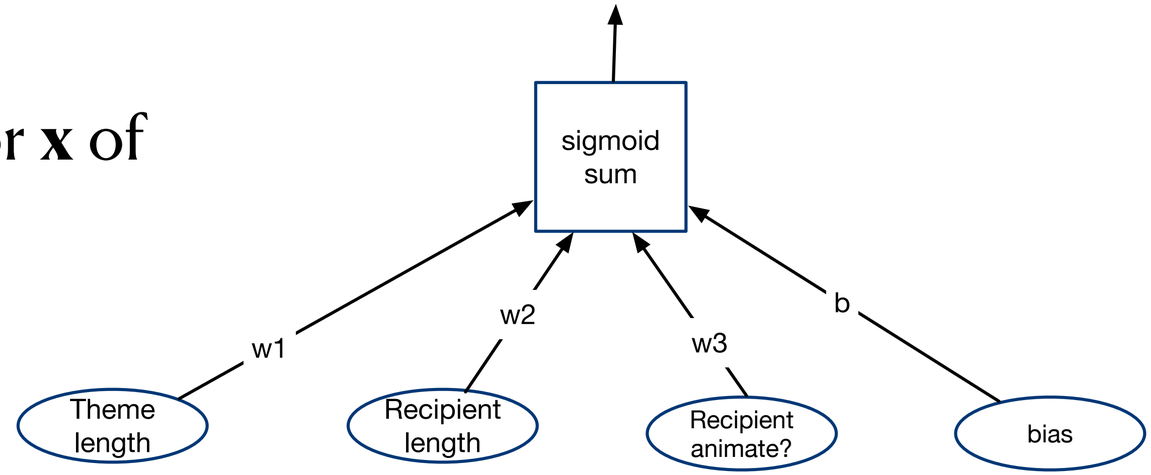


- We can use a similar approach to compute word vectors:
  - Say t is a target word, and w is a potential context word
  - Yes/no question: Does w appear in the corpus as an actual context word of t?

$$P(\text{yes context}|t, w) = \sigma(\text{vec}(t) \cdot \text{vec}(w))$$

Generalizing: logistic regression as a neural network

- Using notation of vector **w** of weights, vector **x** of predictors: P(yes) = sigmoid(**wx**)
- Fitting: For every data point (**x**, y), tweak the weights to make the predicted output a bit more like the true y
- We can use a similar approach to compute word v
  - Say t is a target word, and w is a potential cont
  - Yes/no question: Does w appear in the corpus as an actual context word of t?



vec(t), vec(w) are vectors of weights, just like the **w** for log. regression. We want to learn vec(t), vec(w) from data in the same way

$$P(\text{yes context}|t, w) = \sigma(\text{vec}(t) \cdot \text{vec}(w))$$

Word2Vec SkipGram: Using a neural model to compute word vectors that behave just like count-based vectors

- Every target word t should have a vector vec(t)
- Every context word w should have a vector vec(w)
  - (Let's say that target vectors and context vectors don't have to be the same)
- For w's that are **actual context words of t,** we want the **dot product** vec(t) * vec(w) to be **high**
  - Dot product: the numerator of cosine similarity, $\text{vec}(t) \cdot \text{vec}(w) = \sum_i t_i \cdot w_i$
- For w's that we've **never seen as contexts** of t, we want the dot product vec(t) * vec(w) to be **low**

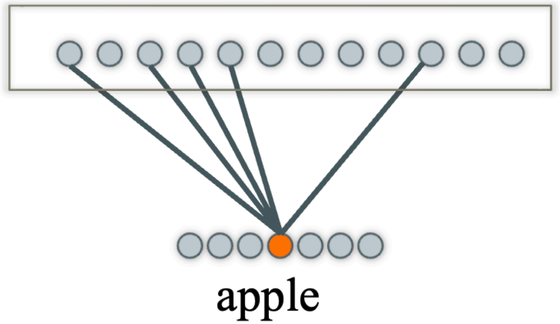
Word2vec SkipGram: Learning word vectors

- Learning from data:
  - Positive data: our corpus.
    - Target word = apricot
    - corpus contains “put some apricot jam on top”
    - “jam” is observed context word for “apricot”
  - Negative data: random target/context pairs, like “apricot/consideration”

$$P(\text{yes context}|t, w) = \sigma(\text{vec}(t) \cdot \text{vec}(w))$$

- Start with random target word embeddings vec(t), and random context word embeddings vec(w)
- For every datapoint (t, w):
  - Predict probability that w is a real context word of t
  - Use loss function to determine how far off we are
  - Adjust all weights a little
- Repeat

first step of the model: mapping “apple” to a whole vector of values



Word2vec SkipGram vectors

- One vector per word
- **Very similar to count-based vectors:**
  - Words that are similar in meaning will have similar target vectors with high **cosine similarity**
  - We can determine **nearest neighbors** like for count-based vectors
  - We can down-project the space to 2 dimensions and **plot the neighborhood**, just for count-based vectors
- **What is different?**
  - Dimensions don't stand for individual context words, they can't be interpreted
    - But then, dimension labels weren't super useful for count-based vectors either

Prediction-based word vectors

- Word2vec, in 2 variants, CBOW and SkipGram, which differ in details of the training
- GLoVE
- fasttext

- There doesn't seem to be a clear advantage, across tasks, for one type of vectors over another. But the training data, and the dimensionality of the vectors, matters:
  - More training data, and larger vectors, generally make better models
  - Varied data, as in Wikipedia, seems better than all news articles

Word token embeddings

Vectors / embeddings for words in context

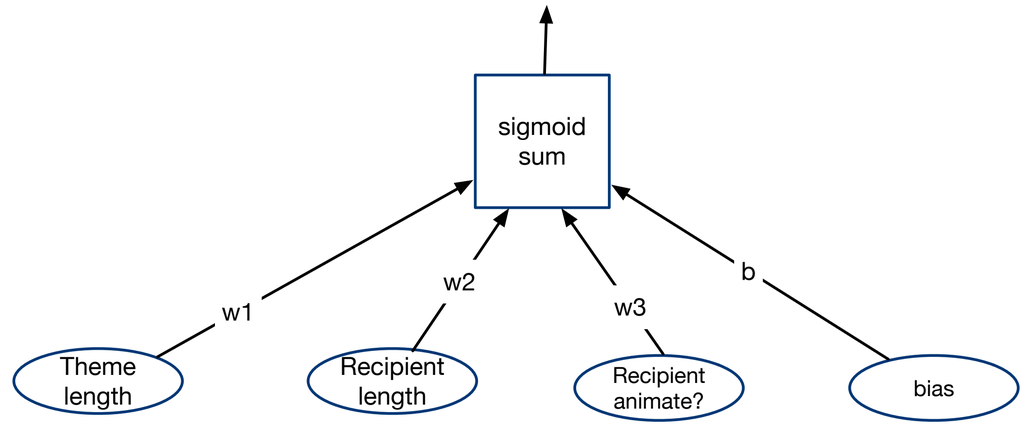
Previously: One vector for the word “bass”, including contexts for all its senses:
the fish, the instrument, the human voice range

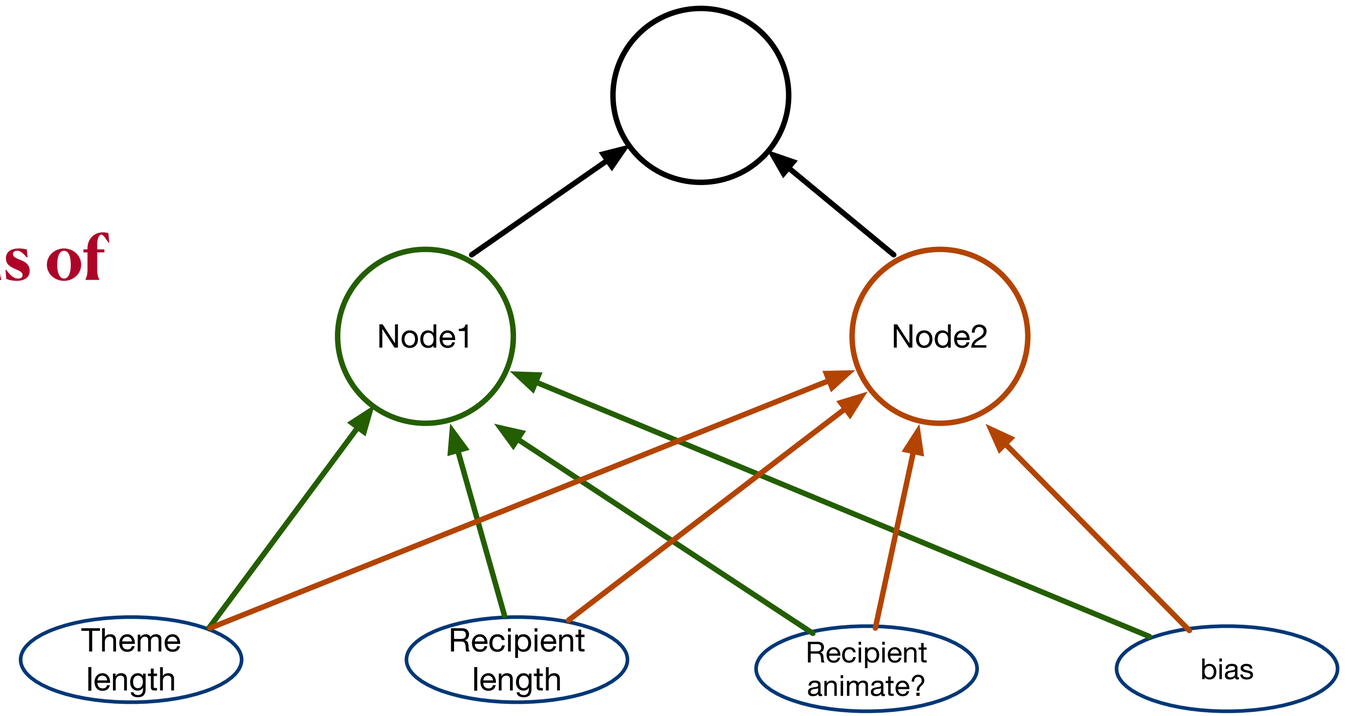
Now we can also obtain a vector for each occurrence of the word “bass”, **word token embeddings**:
(Examples from the British National Corpus)

- Drums and *bass* were as solid as they come and the keyboards filled in any gaps.
- I do like to be able to pick a fish up by its head,’ William Black said, waving a sea *bass* in rigor mortis at me, before going on to inspect its gills.
- As a *bass*, his appearances in the choir at the Three Choirs Festivals found a singer who ‘could really get the bottom notes’.
- There is nothing worse than playing a *bass* guitar — without an amp because you can't afford one yet — all on your jack up in your bedroom.

Building up larger networks from smaller ones

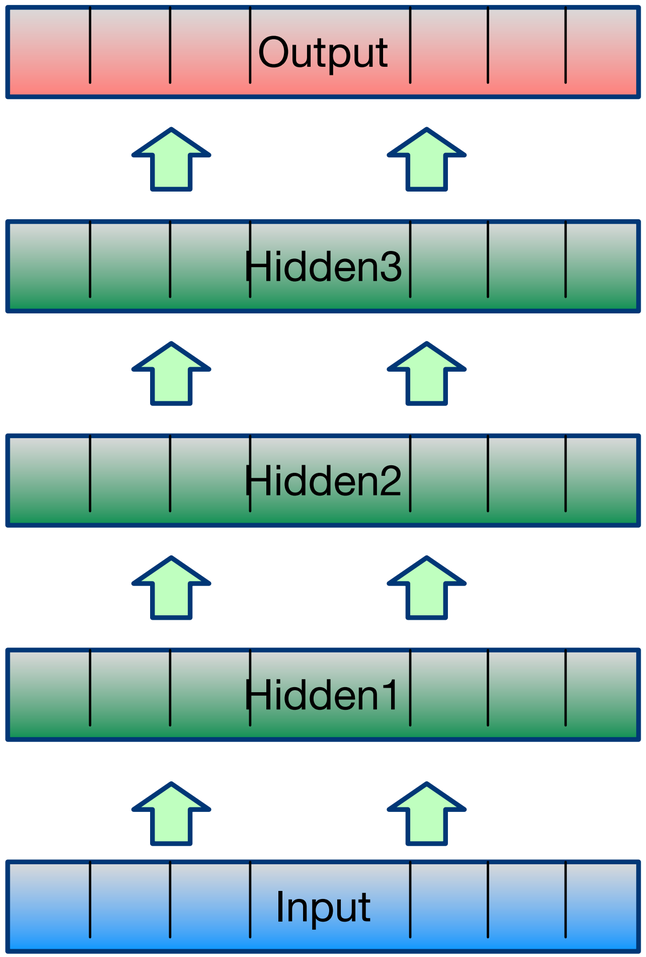
- Green: a logistic regression model
- Red: another logistic regression model
- Larger model learns to combine predictors into internal features
- New **hidden layer**
- **Network learns combinations of inputs to match the task: It can learn its own features!**



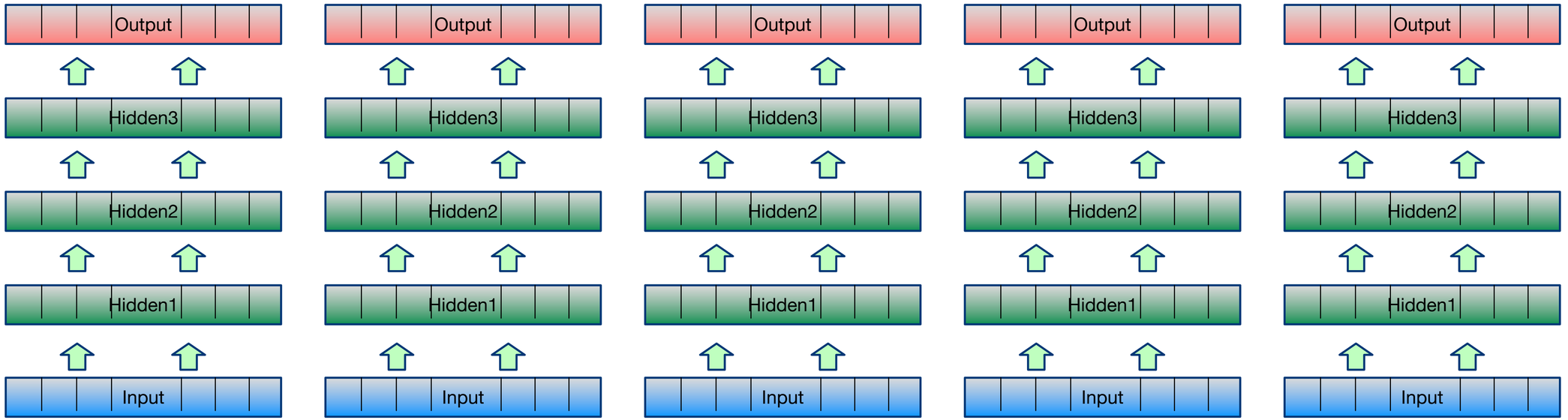


We can build a network with more hidden layers

Deep learning is called Deep Learning because it uses neural networks with many hidden layers

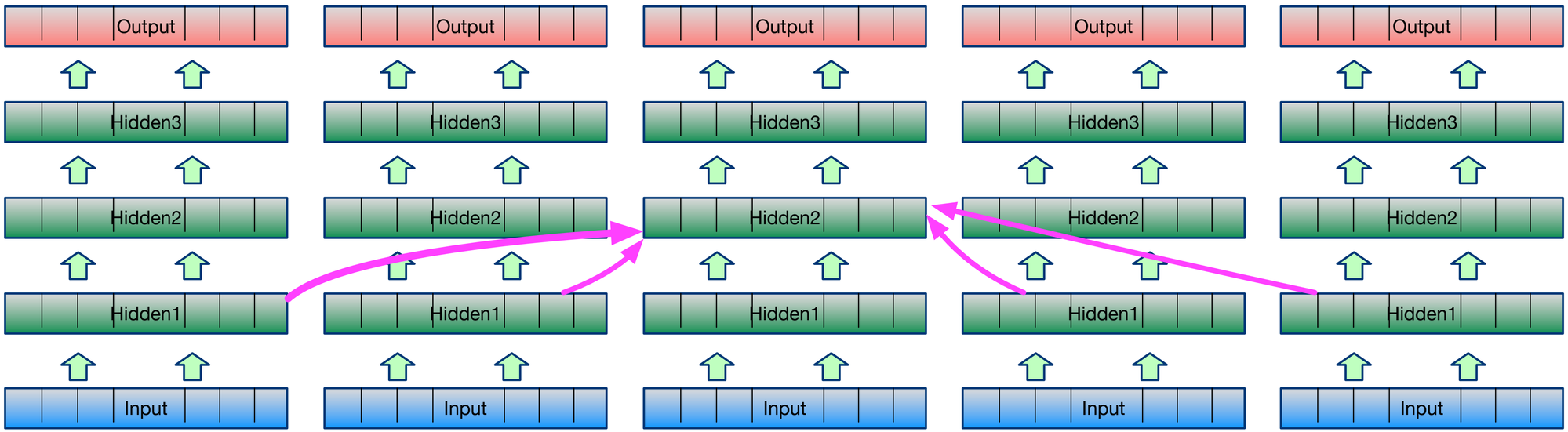


Transformers



Transformers read several input words at once, and transform each of them through several layers.
Each green box is a separate embedding.

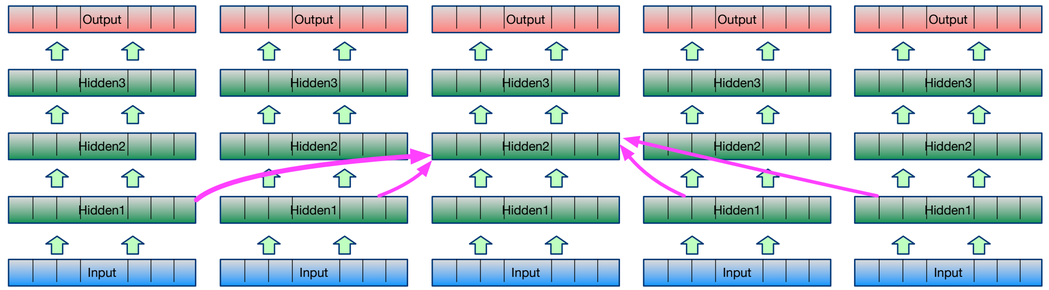
Transformers

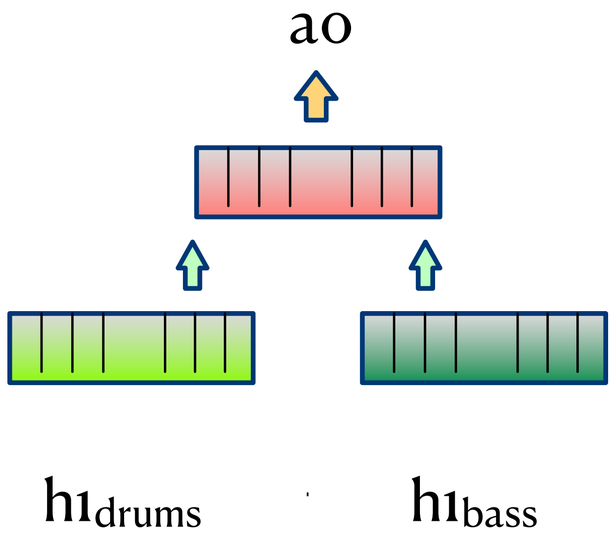


Layer 2 representation for “bass”: based on layer 1 representation for “bass”,
plus weighted sum of layer-1 representations for all other worrds
“**Attention weights**” are computed based on how relevant each other word is for meaning of “bass”

Attention weights

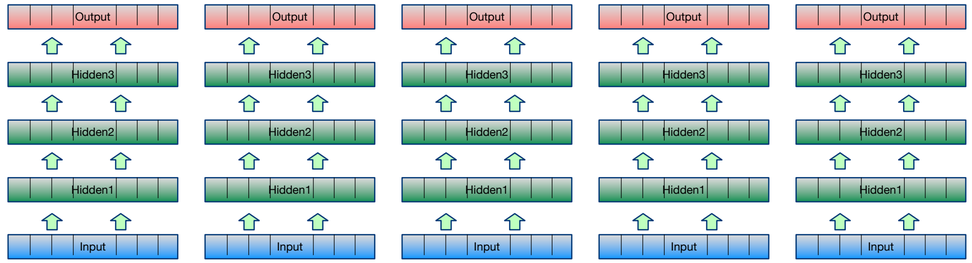
- Say $h1_{bass}$ is the first-level embedding for “bass”
- $h2_{bass} = f(h1_{bass}, a0 * h1_{drums} + a1 * h1_{and} + a2 * h1_{were} + a3 * h1_{as} + ...)$
- a0: attention weight, how important is $h1_{drums}$ for determining the next embedding for “bass”?
- Build a little network to compute this: This again has weights/parameters. Train them as part of the overall training of the whole model.





One large language model: BERT

- Transformer model
- Trained on **masked word task:** "Drums and bass were as [MASK] as they come"
  Guess which word I'm masking.
- "drum" gets many embeddings, one at each layer.
  At layer >1, embedding is influenced by this
  particular sentence context, both left and right
- Also trained on sentence pair task: given sentences s1, s2,
  does s2 come directly after s1 in the corpus?



Large Language Models (LLMs) in general

- **Important difference from count-based models, Word2vec etc: Pre-training**
  - Pre-trained on masked prediction task on large amount of data
  - What you download: weights from pre-training, not pre-computed vectors
  - Apply the pre-trained model to your data to obtain embeddings for the word tokens, specific to this context
  - Big difference:
    - Word type model specific to your data: train a model from scratch on your data.
      may not be good if you have little data
    - Using a LLM: Pre-train on tons of data, then apply to your data.
    - Do you get good performance even when you have little data?
      Possibly, if the phenomena in your data are also reasonably frequent in the pre-training,
      so that the model is sensitive to them

Large Language Models (LLMs) in general

- LLMs with bidirectional input, focus on producing embeddings, e.g.:
  - BERT-base: 11 layers. BERT-large: 23 layers
  - RoBERTa
  - ...
- LLMs with left context only, focus on producing text output, e.g.
  - GPT-2, 3,4
  - ...
- Which is best? unclear. I've seen better performance on lexical semantics tasks with BERT than RoBERTa, but there is no clear
- **HuggingFace library: Very easy to switch between different LLMs in your code, best try different ones**