Sparse and dense embeddings as inp

Input to feedforward neural networks

- A bag-of-word model where the input $\mathbf{x}$ is the count of each word (feature) x_i .

  - The connections from word (feature) dden units z_k form a vector $\theta_i^{(x \to z)}$ th bed as the emb

  - With su earned within t

- *Pretrained* word embeddings learn nlabeled data, using techniques such as Word2V

- *Contextualized* word embeddings (e.g., ELMO, BERT) that are computed dynamically for a word sequence. The requires more advanced architectures (Transformers) that we will talk about later in the course.

One-hot encodings for features

A *one-hot* encoding is one in which each dimension corresponds to a unique feature, and the resulting feature vector of a c instance can be thought of as the sum of indicator feat in which a single dime a value of zero.

Example:

When considering a bag-of-words represen lary of 40000 words. A short document of 20 words will be represented with a very sparse 40000-dimensional vector in which **at most** 20 dimensions have non-zero values

Sparse vectors for text classification

Sparse vectors for text classification can be viewed as a summation of one-hot features for a text instance:

$$\begin{bmatrix}0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}$$

$$+$$

$$\begin{bmatrix}0 & 0 & 0 & 0 & 0\end{bmatrix}$$

$$=$$

$$\begin{bmatrix}0 & 1 & 0 & 1 & 0 & 0 & 1 & 0\end{bmatrix}$$

Shortcomings for sparse representations

- Each feature is a sparse vector in which one dimension is 1 and the rest are 0s (thus “one-hot”)
- Dimensionality of one-hot vector is same as n features
- Features can . he feature “wor as it is to “word is ‘cat”’
- Features for one classifying instance ca summation.
- A recent trend is to use dense representations that can capture similarities between features, which lead to better generalizations to new data.

Dense vectors for text classification

- Extract a set of ling $_1$ $_k$ at are relevant for predicting the output class

- For each feature f_i of interest, retrieve the corresponding vector v_i, which can be pre-trained, pre-co initialized.

- Each core feat ensional space (typically 50 ce.

- Combine the vectors (either by concate n, or a combination of both) into an input vecto for a classification instance.

  - Note: concatenation if we care about relative position, but doesn't work for variable-length vectors such as document classification
- Model training will cause similar features to have similar vectors - information is shared between similar features.

Relationship between one-hot and dense vectors

- Dense represent
  pre-trained word embeddings.

- One-hot and dense representations may no
  one might think

- In fact, using sparse, one-hot vectors as input
  neural netwo
  network to lea
  feature] based on training data.

- With task-specific word embedding, t
  typically smaller, but the training objective for the embedding and the task objective are one and same
- With pre-trained word embeddings, the training data is easy to come by (just unannotated text), but the embedding objective and task objective may diverge.

Two ways of obtaining dense word vectors

- Count based methods, known in NLP as Distr Semantic Models (DSM) or Vector Seman
- Predictive m rk community ntations for words, co
  - Distributed word representations uct of neural language models and later bec its own

Distributional semantics

- Based on the well-known observation of Z. Harris: Words are similar if they occur in the same context (Harri
- Further summarized it as a slogan: "You shall the company it keeps." (J. R. Firth, 1957)
- A long history o nt word meanin represents a context word it can occur wit
- Each word is represented as a sparse vecto high-dimensional space
- Then word distances and similarities can be computed with such a matrix

Steps for building a distributional semantic model

- Preprocess a (large) corpus: tokenization at a minimum, possibly lemmatization, POS tagging, or sy
- Define the "context" for a target term (words. The context can be a window centered on the target term, terms that are s (subject-of,
- Compute a term-context matrix where ds to a term and each column corresponds to a c the target term.
- Each target term is then represented with a high-dimensional vector of context terms.

Mathematical processing for building a DSM

- Weight the term-context matrix with association strength metrics such as Positive Pointwise Mutual Information (PPMI) to correct frequency bias

$$PPMI(x,y) = max(\text{lo}\ \frac{}{}\ )$$

- Its dimensio tion techniques such as *singular value d* (SVD)

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}$$

$$\boldsymbol{A} \in \mathbb{R}^{m\times n}, \boldsymbol{U} \in \mathbb{R}^{m\times k}, \boldsymbol{\Sigma} \in \mathbb{R}^{k\times k}, \boldsymbol{V} \in \mathbb{R}^{n\times k}, n >> k$$

- This will result in a matrix that has much lower dimension but retains most of the information of the original matrix.

Getting pre-trained word embeddings using predictive methods

- Learns word embeddings from large naturally occurring text, using various language model objectives.

  - Decide on the context window

  - Define the objective function that is used to p
    contex et
    word ba

  - Train the neur
  - The resulting weight matrix will serve
    representation for the target word

- "Don't count, predict!" (Baroni et al, 2014) conducted systematic studies and found predict-based word embeddings outperform count-based embeddings.
- One of popular early word emdeddings are Word2vec embeddings.

Word2vec

- Word2vec is a software package that consists of two main models: CBOW (Continuous Bag of Words

- It popularized the use of distributed represen to neural netw nspired many follow- Net)

- It has it roots in language modeling (the use o context to predict the target word), but gi getting good language models and focuses instead on getting good word embeddings.

Understanding word2vec: A simple CBOW model with only one context word in

- Input $\boldsymbol{x} \in \mathbb{R}^V$ $_k$ and $x_{k'} = 0$ for $k' \neq k$. $\boldsymbol{\Theta} \in \mathbb{R}^{N \times V}$ is the weight matrix from the input layer to the hidden layer. Each column of $\boldsymbol{\Theta}$ is an N-dimensional vector representation ted word of the input layer.
- $\boldsymbol{\Theta'} \in \mathbb{R}^{V \times}$ put layer and $\boldsymbol{u}_{w_j}$ is the j-th row of core o_j for each target word w_j and context w omputed as:

$$o_j = \boldsymbol{u}_{w_j}^\top \boldsymbol{v}_{w_i}$$

- Finally we use softmax to obtain a posterior distribution

$$p(w_j | w_i) = y_j = \frac{exp(o_j)}{\sum_{j'=1}^{V} exp(o_{j'})}$$

where y_j is the output of the j-th unit in the output layer

A simple CBOW model

Computing the hidden layer is just embedding lookup

Hidden layer computa

$\boldsymbol{v}_{w_i} = \boldsymbol{z} = \boldsymbol{\Theta}\boldsymbol{x} =$

$$\begin{bmatrix} 0.1 & 0.3 & 0.5 & & & & & 0.6 \\ 0.2 & 0.5 & 0.8 & & & & & 0.1 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.8 \\ 0.8 \end{bmatrix}$$

Note there is no activation at the hidden layer (or there is a linear activation function), so this is a "degenerate neural network".

Computing the output layer

$$\boldsymbol{o} = \boldsymbol{\Theta}'\boldsymbol{z} \quad \begin{bmatrix} 0.3 & 0.4 & 0.6 \\ 0.7 & 0.1 & 0.6 \\ 0.5 & 0.2 & 0.7 \\ 0.2 & 0.6 & 0.3 \\ & & \\ & & \\ 0.2 & 0.4 & \\ 0.3 & 0.2 & \\ 0.3 & 0.4 & 0.6 \\ 0.3 & 0.5 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.95 \\ 0.91 \\ 0.97 \\ 0.82 \\ 1.18 \\ 0.31 \\ 1.06 \\ 0.39 \\ 0.95 \\ 0.63 \end{bmatrix}$$

Each row of $\boldsymbol{\Theta}'$ correspond to vector for a target word w_j.

Taking the softmax over the output

$$\begin{bmatrix} 0.11039215 \\ 0. \\ 0. \end{bmatrix}$$

$$\begin{bmatrix} 0.11039215 \\ 0.08016116 \end{bmatrix}$$

The output $\boldsymbol{y}$ is a probabilistic distribution over the entire vocabulary.

Input vector and output vector

Since there is no activation function at the hidden layer, the output is really just the dot product of the vector of the in context word and the vector of the output target wor

$$p(w_j|w_i) = y_j = \frac{\exp(o_j)}{\sum_{j'=1}^{V} \exp(o_{j'})} = \frac{\exp(\boldsymbol{u}_{w_j}^{\top} \boldsymbol{v}_{w_i})}{\sum_{j'=1}^{V} \exp(\boldsymbol{u}_{w_{j'}}^{\top} \boldsymbol{v}_{w_i})}$$

where $\boldsymbol{v}_{w_i}$ from $\boldsymbol{\Theta}$ is the **input vector** for word w_i and $\boldsymbol{u}_{w_j}$ from $\boldsymbol{\Theta'}$ is the **output vector** for word w_j

Computing the gradient on the hidden-output weights

- Use the familiar cr

$$\ell = -\sum_{j}^{|V|} t_j \log y_j = -\log y_{j\star}$$

where $j\star$ is the index of the target word

- Given y_j is t t on the output is:

$$\frac{\partial \ell}{\partial o_j} = y_j - t_j$$

$$\frac{\partial \ell}{\partial \theta'_{ji}} = \frac{\partial \ell}{\partial o_j}\frac{\partial o_j}{\partial \theta'_{ji}} = (y_j - t_j)z_i$$

- Update the hidden$\rightarrow$output weights

$$\theta'_{ji} = \theta'_{ji} - \eta(y_j - t_j)z_i$$

Updating input→hidden weights

- Compute the erro

$$\frac{\partial \ell}{\partial z_i} = \sum_{j=1}^{V} \frac{\partial \ell}{\partial o_j}\frac{\partial o_j}{\partial z_i} = \sum_{j=1}^{V} (y_j - t_j)\theta'_{ji}$$

- Since

V

The derivative of ℓ on the input s:

$$\frac{\partial \ell}{\partial \theta_{ik}} = \frac{\partial \ell}{\partial z_i}\frac{\partial z_i}{\partial \theta_{ik}} = \sum_{j=1} (y_j - t_j)\theta'_{ji} x_k$$

- Update the input→hidden weights

$$\theta_{ki} = \theta_{ki} - \eta \sum_{j=1}^{V} (y_j - t_j)\theta'_{ij} x_k$$

Gradient computation in matrix form

$$
\boldsymbol{D_o} = \boldsymbol{Y} - \quad = \begin{bmatrix} 0.11039215 \\ 0.10506362 \\ 0.11262222 \end{bmatrix} \begin{matrix} 9215 \\ 6362 \\ 37778 \\ 0.09693485 \\ 0.13893957 \end{matrix}
$$

$$
\begin{bmatrix} 0.05820895 \\ 0.12322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 5820895 \\ 2322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix}
$$

Computing the errors at the hidden layer

$$
\boldsymbol{D_z} = \boldsymbol{D_o}^\top \boldsymbol{\Theta}' =
$$

$$
\begin{bmatrix} 0.110 & 0.106 & -0.887 & 0.097 & 0.139 & 0.0[illegible] & [illegible] & [illegible] & [illegible].110 & 0.080 \end{bmatrix}
$$

$$
\times \begin{bmatrix} 0.3 & 0.4 & 0.6 \\ 0.7 & 0.1 & 0.6 \\ 0.5 & 0.2 & 0.7 \\ 0.2 & 0.6 & 0.3 \\ 0.6 & 0.5 & 0.6 \\ 0.3 & 0.1 & 0.1 \\ 0.2 & 0.4 & 0.8 \\ 0.3 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.6 \\ 0.3 & 0.5 & 0.1 \end{bmatrix} = \Big[-0.115 \quad 0.157
$$

Computing the updates to $\boldsymbol{\Theta}'$

$$\nabla_{\boldsymbol{\Theta}'} = \boldsymbol{D_o z}^\top =$$

$$\begin{bmatrix} 0.110 \\ 0.106 \\ -0.887 \\ 0.097 \\ 0.139 \\ 0.058 \\ 0.123 \\ 0.063 \\ 0.110 \\ 0.080 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.8 & 0.8 \end{bmatrix} = \begin{bmatrix} 00 & 0 & .088 \\ 0.053 & 0.085 & 0.085 \\ [illegible] & 10 & -0.710 \\ [illegible] & 8 & .0778 \\ [illegible] & [illegible] & .111 \\ [illegible] & [illegible] & .0467 \\ [illegible] & [illegible] & .099 \\ 0.032 & 0.050 & 0.050 \\ 0.055 & 0.088 & 0.088 \\ 0.040 & 0.064 & 0.064 \end{bmatrix}$$

Computing the update to $\boldsymbol{\Theta}$

$$\nabla_{\boldsymbol{\Theta}} = \boldsymbol{x}\boldsymbol{D} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} -0.11538456 & 0.1\text{[illegible]} & \text{[illegible]}88611 \end{bmatrix}$$

$$= \begin{bmatrix} 0. & 0. & 0. \\ 0. & 0. & 0. \\ -0.11538456 & 0.15687943 & -0.19388611 \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \end{bmatrix}$$

CBOW for multiple context words

$$\boldsymbol{z} = \frac{}{M}\boldsymbol{\Theta}(\boldsymbol{x}_1 + \boldsymbol{x}_2 + \cdots \boldsymbol{x}_M)$$

$$= \frac{1}{M}(\boldsymbol{v}_{w_1} + \boldsymbol{v}_{w_2} + \cdot$$

where M is the number of words in the context, $_1$ $_2, \cdots, w_M$ are the words in the co r. The loss function is

$$\ell = -\log p(w_j | w_1,$$

$$= -o_{j^\star} + \log \sum_{j'=1}^{V} \exp(o_{j'})$$

$$= -\boldsymbol{u}_{w_j}^\top \boldsymbol{z} + \log \sum_{j'=1}^{V} \exp(\boldsymbol{u}_{w_{j'}}^\top \boldsymbol{z})$$

Computing the hidden layer for multiple context words

$$z = \Theta x =$$

$$\begin{bmatrix} 0.1 & 0.3 & 0.5 & & & & & & & 0.6 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & & & \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & & & \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 3.0 \\ 3.0 \end{bmatrix}$$

During backprop, update vectors for four words instead of just one.

Skip-gram: model

$$p(w_{c,j} = w_{O,c} | w_I) = y_{c,j} = \overline{}$$

where $w_{c,j}$ is the put layer, $w_{O,c}$ is the actual w_I is the only input word, $y_{c,j}$ is the output of the c-th panel of the output layer, $o_{c,j}$ is the net i j-th unit on the c-th panel of the output layer.

$$o_{c,j} = o_j = \boldsymbol{u}_{w_j} \cdot \boldsymbol{z}, \ \textit{for}\ c = 1, 2, \cdots, C$$

Skip-gram: loss function

$$\ell = -\log p(w_{O,1}, w_{O,2}, \cdot \quad , w \quad w \ )$$

$$= -\log \prod^{C} \frac{\exp(o_{c,j_c^\star}}{V}$$

$$= - \quad o_{c,j_c^\star} + C \times \qquad)$$

$$c=1$$

where $j_c^\star$ is the index of the actual c-th output context word.

Combine the loss of C context words with multiplication. Note: $o_{j'}$ is the same for all C panels

Skip-gram: updating the weights

- We take the deriva t of every unit on every p $_{c,j}$, and obtain

$$e_{c,j} = \frac{\partial \ell}{\partial o_{c,j}} = y$$

  which is the prediction error of the unit.

- We define a $\ldots, E_V$ as the sum of the predicti $E_j = \sum_{c=1}^{C} e_{c,j}$

$$\frac{\partial \ell}{\partial \theta'_{ji}} = \sum_{c=1}^{C} \frac{\partial \ell}{\partial o_{c,j}} \cdot \frac{}{\partial \theta'_{ji}} = E_j \cdot z_i$$

- Updating the hidden→output weight matrix:

$$\theta'_{ji} = \theta'_{ji} - \eta \cdot E_j \cdot z_i$$

- No change in how the input→hidden weights are updated.

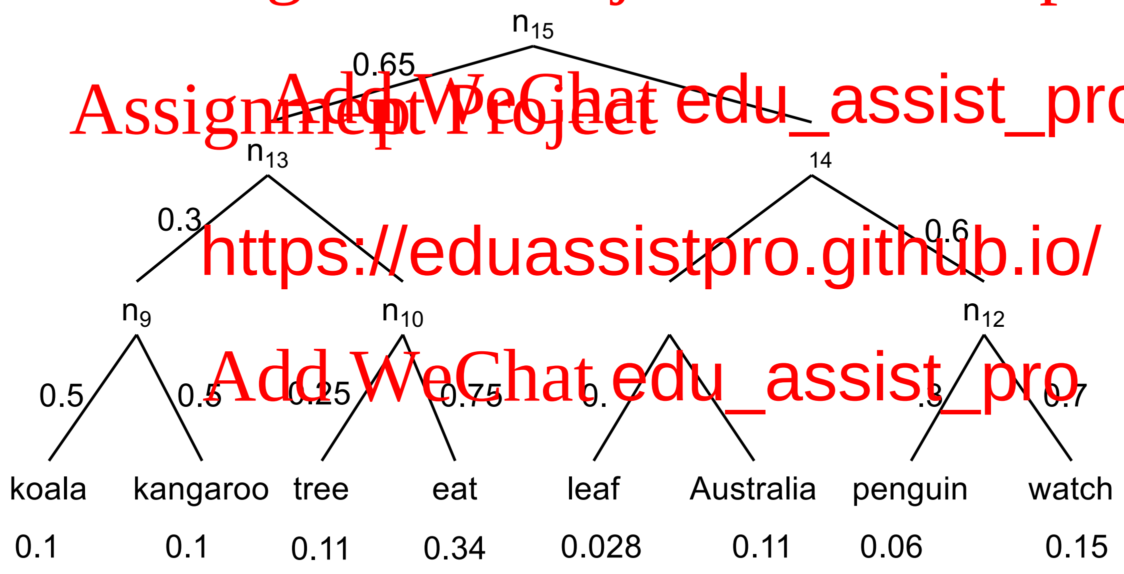
Additional sources on the skip-gram model

- For step-by-step derivation of the Skip-gra excellent tut https://ae demystifying_neural_network_in_ _ _ age_modeling

Optimizing computational efficiency

- Computing softmax at the output layer is exp involves iterative over the entire vocabular
- Two method y
  - **Hierar** ute the probab plexity from $|V|$ to $\log |V|$.
  - **Negative sampling**: Instead of up r all the words in the vocabulary, only sample a small number of words that are not actual context words in the training corpus

Hierarchical softmax



Computing the probabilities of the leaf nodes

$$P(\text{"Kangaroo"}|z) = P_{n_{[illegible]}}(Left|\mathbf{z}) \times P_{n_{[illegible]}}(Right|\mathbf{z})$$

$$P_n(Right|\ \ ) = 1$$

$$P_n(Left|z) = \sigma(\gamma$$

where γ_n is a vector from a set of new parameters that replace $\mathbf{\Theta}$

Huffman Tree Building

A simple algorithm:

- Prepare a collection of n initial Huff ich is a single leaf node. Put the n tre ue organized by weight (frequency).

- Remove the fi oin these two tree trees as children, and whose weight is the su of the two children trees. Put this new tree in queue.

- Repeat steps 2-3 until all of the partial Huffman trees have been combined into one.

Negative sampling

- Computing softmax over the vocabulary is expensive. Another alternative is to approximate softmax by onl sample of (context) words at a time.

- Given a pair of words (w, c), let P e the probability o us, and $P(D =$ ot come from the corpus.

- This probability can be modeled as a sigm

$$P(D = 1|w, c) = \sigma(\boldsymbol{u}_w{}^{\top}\boldsymbol{v}_c) = \frac{1}{1 + e^{-\boldsymbol{u}_w{}^{\top}\boldsymbol{v}_c}}$$

New learning objective for negative sampling

- We need a new obje
  minimize the follo

$$\mathcal{L} = -\sum_{w_j \in D} \log \sigma(o_{w_j}) - \quad \log \sigma(-o_{w_j})$$

  where D is a set of correct context - target wor D' is a set of incorre

- Note that we us
  negative samples. In the skip-gram algo
  multiple positive context words in the ou
  algorithm, there will be only one positive target word.
- The derivative of the loss function with respect to the output word will be:

$$\frac{\partial \ell}{\partial o_{w_j}} = \sigma(o_{w_j}) - t_{w_j}$$

  where $t_{w_j} = 1$ if $w_j \in D$ and $t_{w_j} = 0$ if $w_j \in D'$

Updates to the hidden output weights

- Compute the gradient on the output weights

$$\frac{\partial \ell}{}$$

- When updating the output weights, onl
for words in the positive sample and negat
be updated:

Updates to the input hidden weights

- Computing the derivative of the loss function with respect to the hidden layer

$$\frac{\partial \mathcal{L}}{} = \qquad (\sigma(o_w \qquad {}_w \quad {}_{w_j}$$

- In the CBOW algorithm, the weights for a words will be updated. In the Skip-gram al t for the target word will be updated.

$$\boldsymbol{v}_{w_i} = \boldsymbol{v}_{w_i} - \eta(\sigma(o_{w_j}) - t_{w_j})\boldsymbol{u}_{w_j} x_i$$

How to pick the negative samples?

- If we just randomly
  of any given word w_i getting picked is:

$$p(w_i) = \frac{freq}{\sum_{j=0}^{V} f}$$

  More freque
  may not be idea

- Adjust the formula to give the less frequen
  chance to get picked:

$$p(w_i) = \frac{freq(w_i)^{\frac{3}{4}}}{\sum_{j=0}^{V} freq(w_j)^{\frac{3}{4}}}$$

- Generate a sequence of words using the adjusted probability, and randomly pick $n_{D'}$ words

Use of embeddings: word and short document similarity

- Word embeddings can be used to compute word similarity with cosine similarity

$$sim_{cos} = \frac{\boldsymbol{u}}{\boldsymbol{u}}$$

- How accurat ity can also be use

- They can also be used to compute the simila documents

$$sim_{doc}(D_1, D_2) = \sum_{i=1}^{m} \sum_{j=1}^{n} cos(\boldsymbol{w}_i^1, \boldsymbol{w}_j^2)$$

Use of embeddings: word analogy

- What's even more impressive is that they can be used to compute word analogy

$$analogy(m : w \rightarrow k : ?) = \operatorname*{argmax}_{v \in V \setminus m,k,w} cos(\boldsymbol{v}, \boldsymbol{k} - \boldsymbol{m} + \boldsymbol{w})$$

$$analogy(m : w \rightarrow k : ?) = \operatorname*{argmax}_{v \in V \setminus m,k,w} cos(\boldsymbol{v}, \boldsymbol{k}) - cos(\boldsymbol{v}, \boldsymbol{m}) + cos(\boldsymbol{v}, \boldsymbol{w})$$

$$analogy(m : w \rightarrow k : ?) = \operatorname*{argmax}_{v \in V \setminus m,k,w} \frac{cos(\boldsymbol{v}, \boldsymbol{k}) cos(\boldsymbol{v}, \boldsymbol{w})}{cos(\boldsymbol{v}, \boldsymbol{m}) + \epsilon}$$

Word analogy

https://eduassistpro.github.io/
Add WeChat edu_assist_pro

Use of word embeddings

- Computing word similarities is not a "real" pr eyes of many

- The most important use word embeddings is as input to predict the ou tions

- Many follow embeddings, e.g., GLOVE

  - word2vec: http://vectors.nlpl
  - fasttext: https://fasttext.cc/docs/en/english-vectors.html
  - GLOVE: https://nlp.stanford.edu/projects/glove

Shortcoming of “per-type” word embeddings

- “Per-type” wor 2vec do not account for t
    - “Work out the **solution** in your head.”
    - “Heat the **solution** to 75° Celsius.”

    - Having the same embedding for both insta doesn’t make sense.

- The solution is Contextualized word embed generated on t same word wil different sentences.

    - ELMO: https://allennlp.org/
    - BERT: https://github.com/g rt
    - Roberta: https://pytorch.org/hub/pytorch_fairseq_roberta

- The contextualized word embeddings can be fine-tuned when used in a new classification task in a process called *transfer learning*.
- This turns out to be a very powerful idea that leads to many breakthroughs.

Embeddings in Pytorch