CS388: Natural Language Processing

Lecture 7: Word Embeddings

Greg Durrett

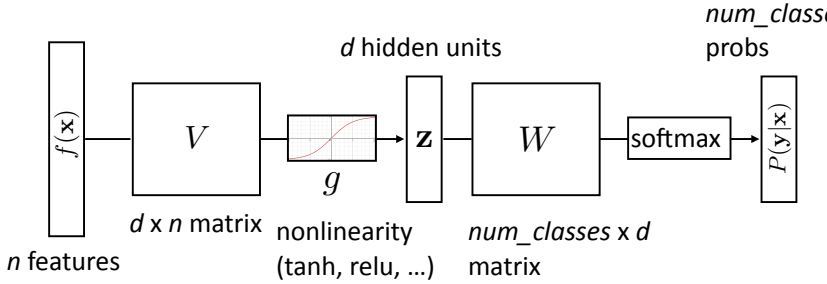
TEXAS
The University of Texas at Austin

Administrivia

- Mini 1 grades out soon
- Project 1 due Thursday

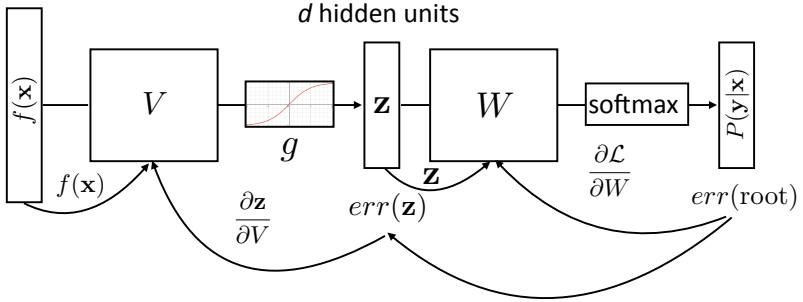
Recall: Feedforward NNs

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



Recall: Backpropagation

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



This Lecture

- Implementing NNs
- Training tips
- Word representations
- word2vec/GloVe
- Evaluating word embeddings

Implementing NNs

(see ffnn_example.py on the course website)

Computation Graphs

- Computing gradients is hard! Computation graph abstraction allows us to define a computation symbolically and will do this for us
- Automatic differentiation: keep track of derivatives / be able to backpropagate through each function:

```
y = x * x  ⟶ (y,dy) = (x * x, 2 * x * dx)
```
codegen

- Use a library like Pytorch or Tensorflow. This class: Pytorch

Computation Graphs in Pytorch

- Define forward pass for $P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$

```
class FFNN(nn.Module):
    def __init__(self, inp, hid, out):
        super(FFNN, self).__init__()
        self.V = nn.Linear(inp, hid)
        self.g = nn.Tanh()
        self.W = nn.Linear(hid, out)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

Computation Graphs in Pytorch

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$

ei*: one-hot vector of the label (e.g., [0, 1, 0])

```
ffnn = FFNN()
def make_update(input, gold_label):
   ffnn.zero_grad() # clear gradient variables
   probs = ffnn.forward(input)
   loss = torch.neg(torch.log(probs)).dot(gold_label)
   loss.backward()
   optimizer.step()
```

Training a Model

Define a computation graph

For each epoch:

For each batch of data:

Compute loss on batch

Autograd to compute gradients

Take step with optimizer

Decode test set

Training Tips

Batching

- Batching data gives speedups due to more efficient matrix operations
- Need to make the computation graph process a batch at the same time

```
# input is [batch_size, num_feats]
# gold_label is [batch_size, num_classes]
def make_update(input, gold_label)
    ...
    probs = ffnn.forward(input) # [batch_size, num_classes]
    loss = torch.sum(torch.neg(torch.log(probs)).dot(gold_label))
    ...
```

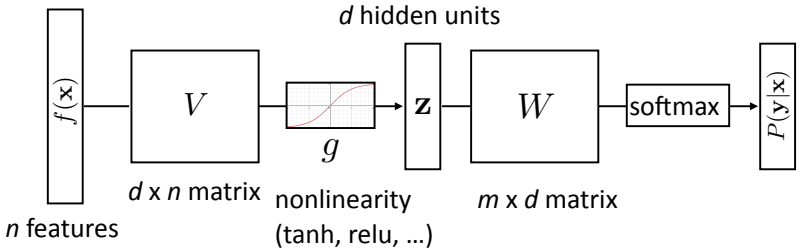
- Batch sizes from 1-100 often work well

Training Basics

- Basic formula: compute gradients on batch, use first-order optimization method (SGD, Adagrad, etc.)
- How to initialize? How to regularize? What optimizer to use?
- This lecture: some practical tricks. Take deep learning or optimization courses to understand this further

How does initialization affect learning?

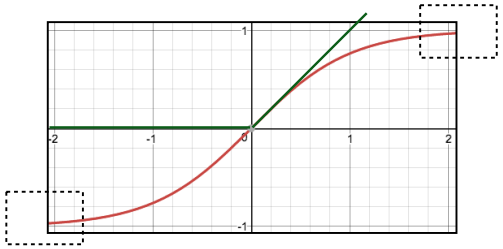
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



- How do we initialize V and W? What consequences does this have?
- *Nonconvex* problem, so initialization matters!

How does initialization affect learning?

- Nonlinear model…how does this affect things?



- If cell activations are too large in absolute value, gradients are small
- ReLU: larger dynamic range (all positive numbers), but can produce big values, can break down if everything is too negative

Initialization

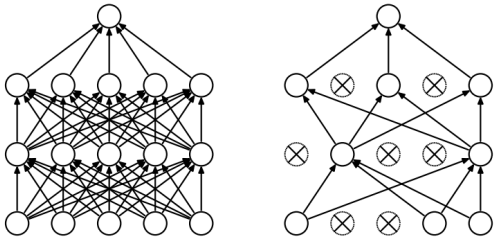
1) Can't use zeroes for parameters to produce hidden layers: all values in that hidden layer are always 0 and have gradients of 0, never change

2) Initialize too large and cells are saturated

- Can do random uniform / normal initialization with appropriate scale
- Glorot initializer: $U\left[-\sqrt{\frac{6}{\text{fan-in}+\text{fan-out}}}, +\sqrt{\frac{6}{\text{fan-in}+\text{fan-out}}}\right]$
  - Want variance of inputs and gradients for each layer to be the same
- Batch normalization (Ioffe and Szegedy, 2015): periodically shift+rescale each layer to have mean 0 and variance 1 over a batch (useful if net is deep)

Dropout

- Probabilistically zero out parts of the network during training to prevent overfitting, use whole network at test time
- Form of stochastic regularization
- Similar to benefits of ensembling: network needs to be robust to missing signals, so it has redundancy
- One line in Pytorch/Tensorflow

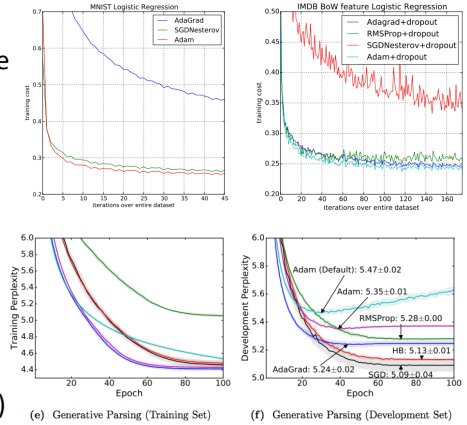


(a) Standard Neural Net (b) After applying dropout.

Srivastava et al. (2014)

Optimizer

- Adam (Kingma and Ba, ICLR 2015): very widely used. Adaptive step size + momentum
- Wilson et al. NIPS 2017: adaptive methods can actually perform badly at test time (Adam is in pink, SGD in black)
- One more trick: **gradient clipping** (set a max value for your gradients)

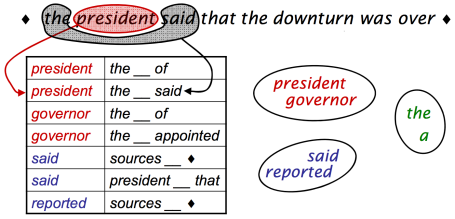


(e) Generative Parsing (Training Set) (f) Generative Parsing (Development Set)

Word Representations

Word Representations

- Neural networks work very well at continuous data, but words are discrete
- Continuous model <-> expects continuous semantics from input
- "You shall know a word by the company it keeps" Firth (1957)



[Finch and Chater 92, Shuetze 93, many others]

slide credit: Dan Klein

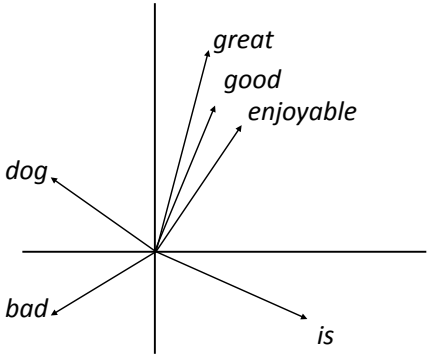
Word Embeddings

- Want a vector space where similar words have similar embeddings

the movie was great

≈

the movie was good



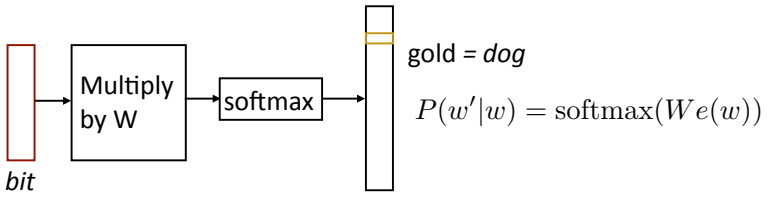
- Goal: come up with a way to produce these embeddings
- For each word, want "medium" dimensional vector (50-300 dims) representing it

word2vec/GloVe

Skip-Gram

- Predict one word of context from word

the dog bit the man



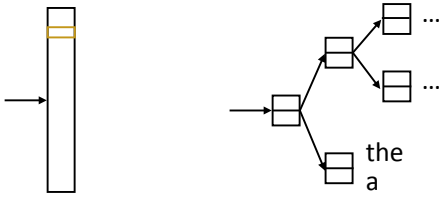
- Another training example: *bit -> the*
- Parameters: *d* x |V| vectors, |V| x *d* output parameters (W) (also usable as vectors!)

Mikolov et al. (2013)

Hierarchical Softmax

$P(w|w_{-1}, w_{+1}) = \text{softmax}\left(W(c(w_{-1}) + c(w_{+1}))\right)$ $P(w'|w) = \text{softmax}(We(w))$

- Matmul + softmax over |V| is very slow to compute for CBOW and SG



- Huffman encode vocabulary, use binary classifiers to decide which branch to take
- log(|V|) binary decisions

- Standard softmax: |V| dot products of size *d*
- Hierarchical softmax: log(|V|) dot products of size *d*, |V| x *d* parameters

Mikolov et al. (2013)

Skip-Gram with Negative Sampling

- Take (word, context) pairs and classify them as "real" or not. Create random negative examples by sampling from unigram distribution

  (*bit, the*) => +1
  (*bit, cat*) => -1
  (*bit, a*) => -1
  (*bit, fish*) => -1

$$P(y=1|w,c) = \frac{e^{w \cdot c}}{e^{w \cdot c}+1}$$

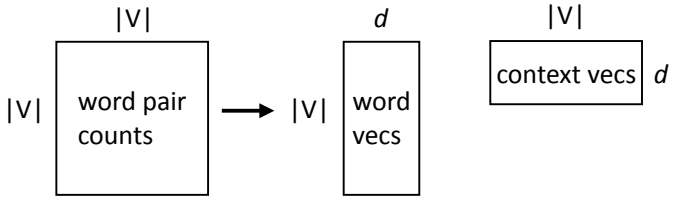
words in similar contexts select for similar *c* vectors

- *d* x |V| vectors, *d* x |V| context vectors (same # of params as before)
- Objective = $\log P(y=1|w,c) + \frac{1}{k}\sum_{i=1}^{n} \log P(y=0|w_i,c)$ (sampled)

Mikolov et al. (2013)

Connections with Matrix Factorization

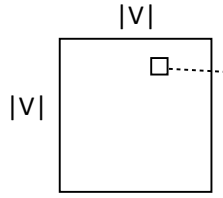
- Skip-gram model looks at word-word co-occurrences and produces two types of vectors



- Looks almost like a matrix factorization…

Levy et al. (2014)

Skip-Gram as Matrix Factorization



num negative samples

$$M_{ij} = \text{PMI}(w_i, c_j) - \log k$$

$$\text{PMI}(w_i, c_j) = \frac{P(w_i, c_j)}{P(w_i)P(c_j)} = \frac{\frac{\text{count}(w_i, c_j)}{D}}{\frac{\text{count}(w_i)}{D}\frac{count(c_j)}{D}}$$

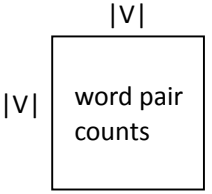
Skip-gram objective *exactly* corresponds to factoring this matrix:

- *If* we sample negative examples from the unigram distribution over words
- …and it's a *weighted* factorization problem (weighted by word freq)

Levy et al. (2014)

GloVe (**Glo**bal **Ve**ctors)

- Also operates on counts matrix, weighted regression on the log co-occurrence matrix



- Objective = $\sum_{i,j} f(\text{count}(w_i, c_j)) \left(w_i^\top c_j + a_i + b_j - \log \text{count}(w_i, c_j)\right)^2$
- Constant in the dataset size (just need counts), quadratic in voc size
- By far the most common word vectors used today (5000+ citations)

Pennington et al. (2014)

fastText: Sub-word Embeddings

- Same as SGNS, but break words down into n-grams with n = 3 to 6

  where:
  3-grams: <wh, whe, her, ere, re>
  4-grams: <whe, wher, here, ere>,
  5-grams: <wher, where, here>,
  6-grams: <where, where>

- Replace $w \cdot c$ in skip-gram computation with $\left(\sum_{g \in \text{ngrams}} w_g \cdot c \right)$

- Advantages?

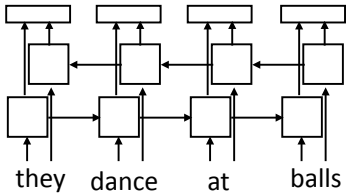
Bojanowski et al. (2017)

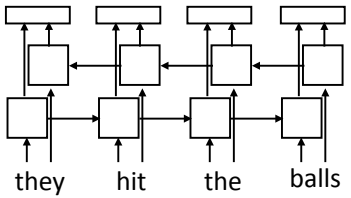
Using Word Embeddings

- Approach 1: learn embeddings as parameters from your data
  - Often works pretty well
- Approach 2: initialize using GloVe, keep fixed
  - Faster because no need to update these parameters
- Approach 3: initialize using GloVe, fine-tune
  - Works best for some tasks

Preview: Context-dependent Embeddings

- How to handle different word senses? One vector for *balls*





- Train a neural language model to predict the next word given previous words in the sentence, use its internal representations as word vectors
- *Context-sensitive* word embeddings: depend on rest of the sentence
- *Huge* improvements across nearly all NLP tasks over GloVe

Peters et al. (2018)

Compositional Semantics

- What if we want embedding representations for whole sentences?
- Skip-*thought* vectors (Kiros et al., 2015), similar to skip-gram generalized to a sentence level (more later)
- Is there a way we can compose vectors to make sentence representations? Summing?
- Will return to this in a few weeks as we move on to syntax and semantics

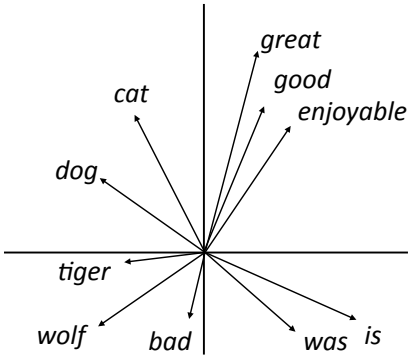
Evaluation

Evaluating Word Embeddings

- What properties of language should word embeddings capture?
- Similarity: similar words are close to each other
- Analogy:

  good is to best as smart is to ???

  Paris is to France as Tokyo is to ???



Similarity

Method	WordSim Similarity	WordSim Relatedness	Bruni et al. MEN	Radinsky et al. M. Turk	Luong et al. Rare Words	Hill et al. SimLex
PPMI	.755	**.697**	.745	.686	.462	.393
SVD	**.793**	.691	**.778**	.666	**.514**	.432
SGNS	**.793**	.685	.774	**.693**	.470	**.438**
GloVe	.725	.604	.729	.632	.403	.398

- SVD = singular value decomposition on PMI matrix
- GloVe does not appear to be the best when experiments are carefully controlled, but it depends on hyperparameters + these distinctions don't matter in practice

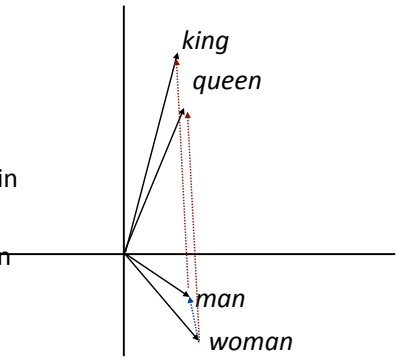
Levy et al. (2015)

Analogies

(king - man) + woman = queen

king + (woman - man) = queen

- Why would this be?
- woman - man captures the difference in the contexts that these occur in
- Dominant change: more "he" with man and "she" with woman — similar to difference between king and queen
- Can evaluate on this as well



What can go wrong with word embeddings?

- What's wrong with learning a word's "meaning" from its usage?
- What data are we learning from?
- What are we going to learn from this data?

What do we mean by bias?

- Identify *she* - *he* axis in word vector space, project words onto this axis

Extreme *she* occupations

1. homemaker	2. nurse	3. receptionist
4. librarian	5. socialite	6. hairdresser
7. nanny	8. bookkeeper	9. stylist
10. housekeeper	11. interior designer	12. guidance counselor

Extreme *he* occupations

1. maestro	2. skipper	3. protege
4. philosopher	5. captain	6. architect
7. financier	8. warrior	9. broadcaster
10. magician	11. figher pilot	12. boss

Bolukbasi et al. (2016)

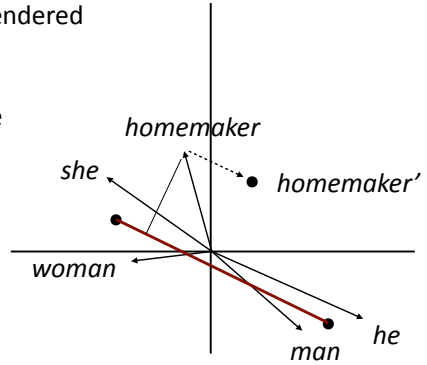
- Nearest neighbor of (b - a + c)

Racial Analogies	
black → homeless	caucasian → servicemen
caucasian → hillbilly	asian → suburban
asian → laborer	black → landowner
Religious Analogies	
jew → greedy	muslim → powerless
christian → familial	muslim → warzone
muslim → uneducated	christian → intellectually

Manzini et al. (2019)

Debiasing

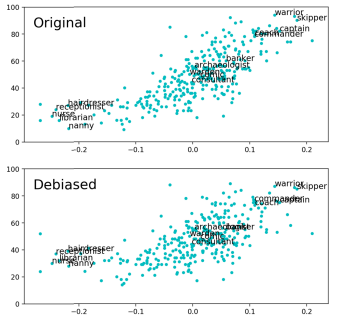
- Identify gender subspace with gendered words
- Project words onto this subspace
- Subtract those projections from the original word



Bolukbasi et al. (2016)

Hardness of Debiasing

- Not that effective…and the male and female words are still clustered together
- Bias pervades the word embedding space and isn't just a local property of a few words



(a) The plots for HARD-DEBIASED embedding, before (top) and after (bottom) debiasing.

Gonen and Goldberg (2019)

Takeaways

- Lots to tune with neural networks
  - Training: optimizer, initializer, regularization (dropout), …
  - Hyperparameters: dimensionality of word embeddings, layers, …
- Word vectors: learning word -> context mappings has given way to matrix factorization approaches (constant in dataset size)
- Lots of pretrained embeddings work well in practice, they capture some desirable properties
- Even better: context-sensitive word embeddings (ELMo)
- Next time: RNNs and CNNs