Word Vectors

CS 780/880 Natural Language Processing Lecture 14

Samuel Carton, University of New Hampshire

Last lecture

Feedforward neural nets

Backpropagation

GPU operations on tensors

Training on GPU

Pytorch Lightning

- LightningModule
- Trainer

Data sparsity

A big problem with everything we've done so far is that our data is **sparse** and the models always **learn from scratch**

- e.g. learning that "idiot" → toxicity doesn't learn that "moron" → toxicity
- e.g. learning that "wonderful" → positive doesn't learn that "great" → positive

This is limiting. It means that models can only learn from what's in front of them and can't leverage basic knowledge of the language.

Also, big sparse count/TFIDF matrices are a pain to work with, computationally

How to fix?

Distributional hypothesis

Basic idea: in a given corpus of text, similar words tend to occur in similar contexts

Examples:

“You are a gigantic [moron|idiot|dumb-dumb].”
“That was a really [moronic|idiotic|dumb] thing to do.”

“It was a [wonderful|great|stupendous] movie.”
“The casting was just [wonderful|great|stupendous].”

How to leverage?

Word vector models

Basic idea: generate a **dense vector representation** of a word that is predictive of the contexts it is likely to occur in.

- Then, similar words will have similar vectors

Basic workflow:

1. Train word vectors on big unlabeled corpus
2. Save as big mapping of word → vector
3. Use these pretrained vectors as starting point for specific tasks
   - Classification
   - Language modeling
   - Translation
   - etc.

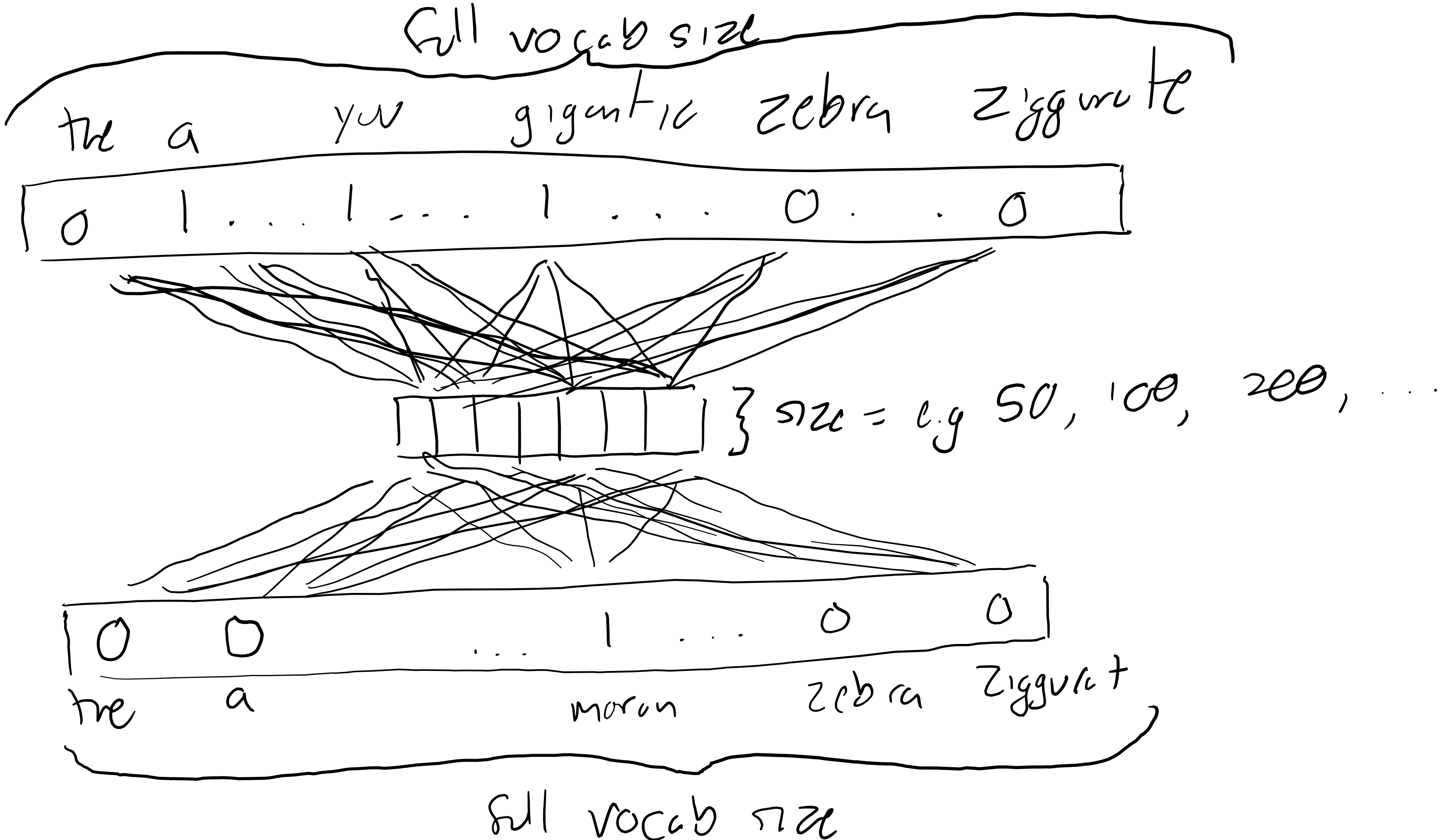
Word2Vec

Mikolov et al. (2013)

Basic idea: Train a feed-forward neural network to take unigram representation of word (i.e. the size of the vocabulary), squish it down to small dimension (e.g. 50), then predict unigram representation of co-occurring words

Word2Vec

“You are a gigantic [moron|idiot|dumb-dumb].”



Word2Vec

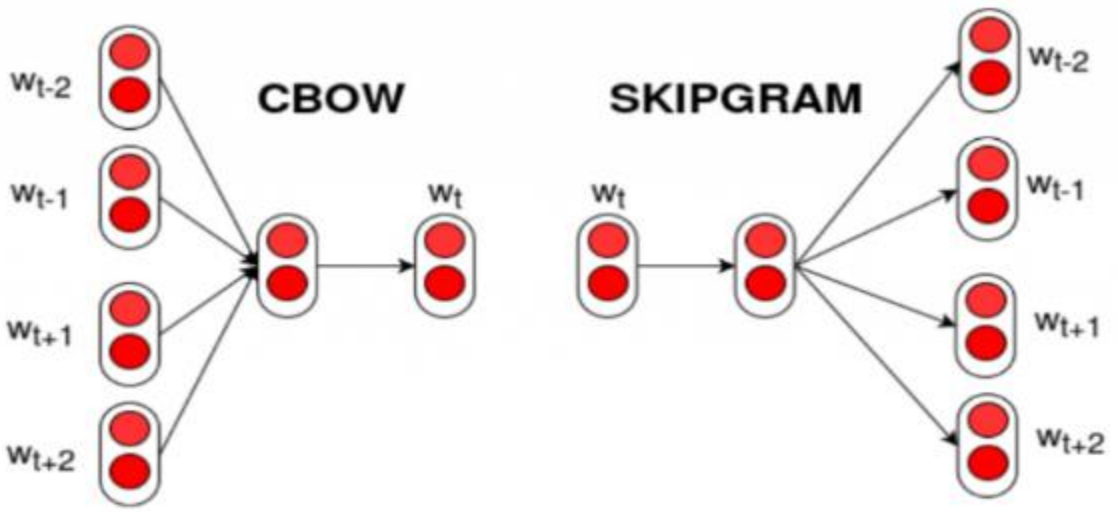
Basic algorithm:

1. Take unlabeled corpus, e.g. **all of Wikipedia**
2. Divide it into a series of (word, context) pairs
3. Choose an embedding size (50, 100, 200, 300, etc.)
4. Train a 2-layer feedforward model with two layers:
   - Encoder: vocab size × embedding size
   - Decoder: embedding size × vocab size
5. Use gradient descent to train model to encode words, then decode to predict context
   - Use cross entropy for loss function
6. When you are done training:
   - Encoder should map similar words to similar intermediate representations
   - Run encoder over entire vocabulary to get a dense vector for each word, then save for later
   - Throw away decoder

Word2Vec: two variants

There are actually two variants of Word2Vec:

- **Continuous bag-of-words (CBOW):** Takes in context, predicts word
  - Faster to train, better for frequent words, I'm told
- **Skip-gram**: Takes in word, predicts context
  - Better for rare words, apparently



How to choose?

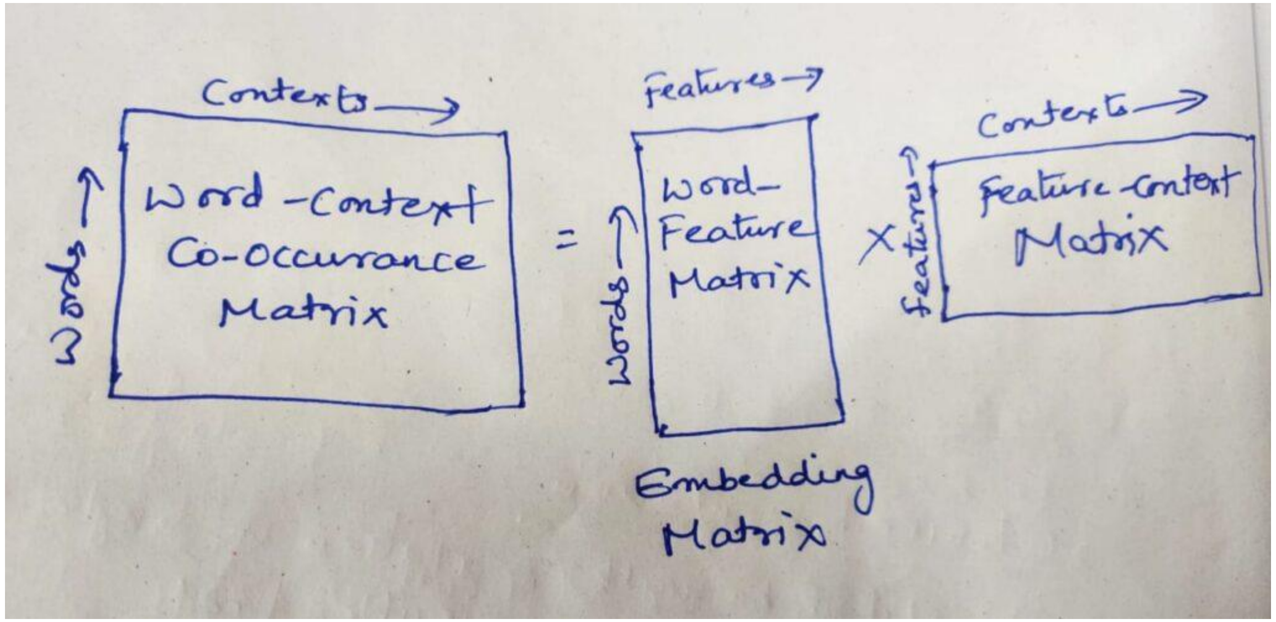
https://machinelearninginterview.com/topics/natural-language-processing/what-is-the-difference-between-word2vec-and-glove

GloVe embeddings

For pretrained embedding vectors, use GloVe instead:

- Pennington et al. (2014), https://nlp.stanford.edu/projects/glove/

Trained by doing matrix factorization of giant N × N word-co-occurrence matrix

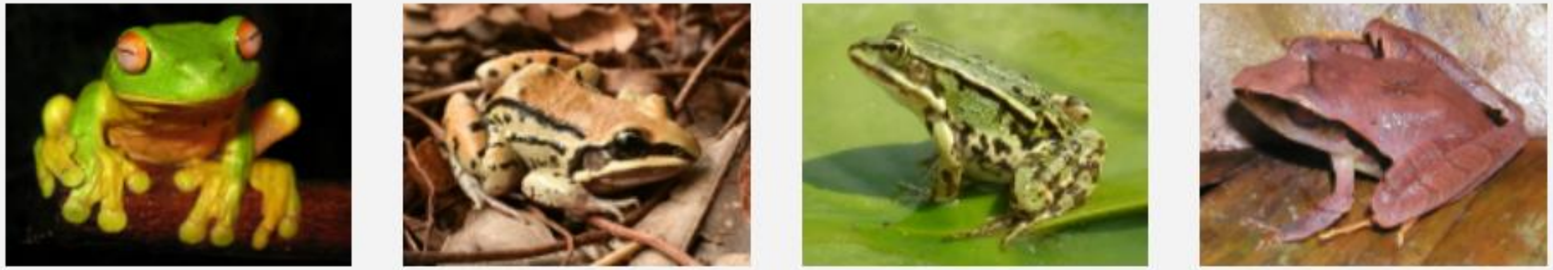


https://machinelearninginterview.com/topics/natural-language-processing/what-is-the-difference-between-word2vec-and-glove

Word vectors capture word similarity

In both GloVe and Word2Vec, similar words will end up with vectors that are close in vector space

0. *frog*
1. frogs
2. toad
3. litoria
4. leptodactylidae
5. rana
6. lizard
7. eleutherodactylus

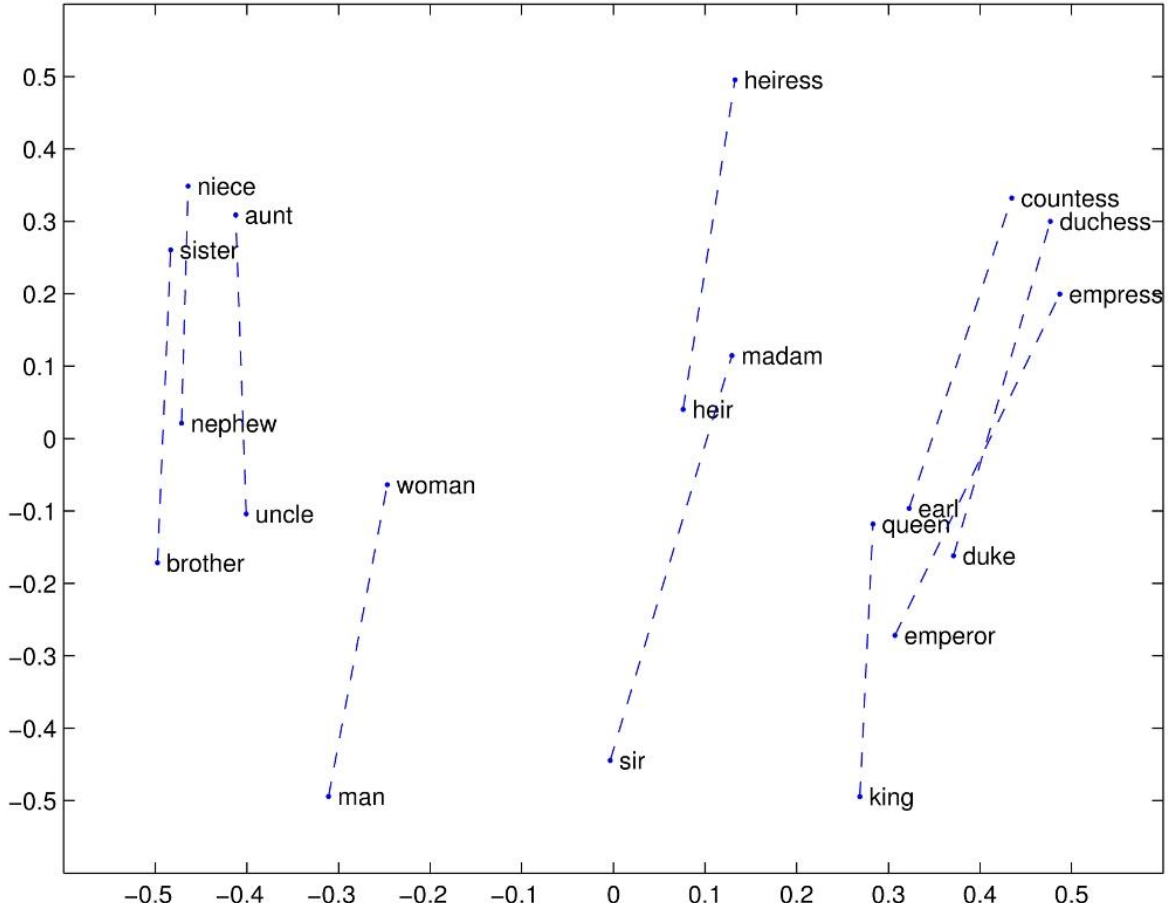


3. litoria 4. leptodactylidae 5. rana 7. eleutherodactylus

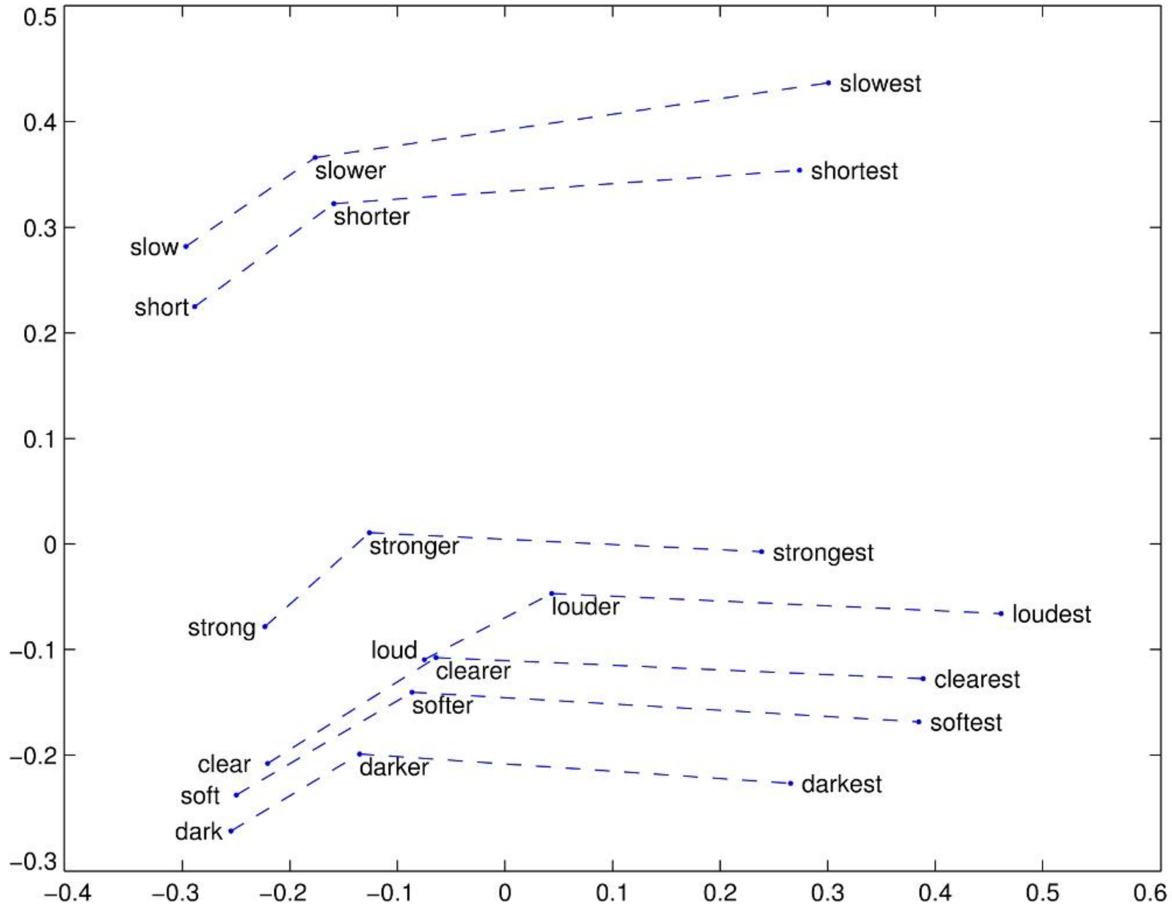
https://nlp.stanford.edu/projects/glove/

Word vectors capture analogy

Gender



Word senses



https://nlp.stanford.edu/projects/glove/

Reading GloVe embeddings

```
glove_url = 'https://github.com/uclnlp/inferbeddings/raw/master/data/glove/glove.6B.50d.txt.gz'

# glove_url = 'https://github.com/allenai/spv2/raw/master/model/glove.6B.100d.txt.gz'

# More available at http://nlp.uoregon.edu/download/embeddings/
```

```
import numpy as np
glove_data = np.loadtxt(glove_url, dtype='str', comments=None)
```

```
print(glove_data)

# The vocab size for this particular embedding is 400,000
print(glove_data.shape)
```

```
[['the' '0.418' '0.24968' ... '-0.18411' '-0.11514' '-0.78581']
 [',' '0.013441' '0.23682' ... '-0.56657' '0.044691' '0.30392']
 ['.' '0.15164' '0.30177' ... '-0.35652' '0.016413' '0.10216']
 ...
 ['rolonda' '-0.51181' '0.058706' ... '-0.25003' '-1.125' '1.5863']
 ['zsombor' '-0.75898' '-0.47426' ... '0.78954' '-0.014116' '0.6448']
 ['sandberger' '0.072617' '-0.51393' ... '-0.18907' '-0.59021' '0.55559']]
(400000, 51)
```

Reading GloVe embeddings

```python
# Split the downloaded data into a vocab list and vector matrix
glove_words = glove_data[:,0]
glove_vectors = glove_data[:,1:].astype('float')

print('Vocabulary:')
print(glove_words)

print('Vectors:')
print(glove_vectors)
```

```
Vocabulary:
['the' ',' '.' ... 'rolonda' 'zsombor' 'sandberger']
Vectors:
[[ 0.418     0.24968  -0.41242  ... -0.18411  -0.11514  -0.78581 ]
 [ 0.013441  0.23682  -0.16899  ... -0.56657   0.044691  0.30392 ]
 [ 0.15164   0.30177  -0.16763  ... -0.35652   0.016413  0.10216 ]
 ...
 [-0.51181   0.058706  1.0913   ... -0.25003  -1.125     1.5863  ]
 [-0.75898  -0.47426   0.4737   ...  0.78954  -0.014116  0.6448  ]
 [ 0.072617 -0.51393   0.4728   ... -0.18907  -0.59021   0.55559 ]]
```

```python
# We will need a vocab index later
glove_vocab = {}
for i, word in enumerate(glove_words):
    glove_vocab[word] = i
```

Properties of GloVe vectors

```python
# Finding the word vectors for a bunch of words I want to look at
kingv = glove_vectors[glove_vocab['king']]
queenv = glove_vectors[glove_vocab['queen']]
personv = glove_vectors[glove_vocab['person']]
presidentv = glove_vectors[glove_vocab['president']]
monarchv = glove_vectors[glove_vocab['monarch']]
ceov = glove_vectors[glove_vocab['ceo']]

doctorv = glove_vectors[glove_vocab['doctor']]
nursev = glove_vectors[glove_vocab['nurse']]


manv = glove_vectors[glove_vocab['man']]
womanv = glove_vectors[glove_vocab['woman']]

moronv = glove_vectors[glove_vocab['moron']]
idiotv = glove_vectors[glove_vocab['idiot']]
geniusv = glove_vectors[glove_vocab['genius']]
prodigyv = glove_vectors[glove_vocab['prodigy']]
```

```python
from scipy.spatial.distance import cosine as cosdis
```

Properties of GloVe vectors

```
# Words with similar meanings tend to have closer vectors than words with opposite meanings
print('Moron vs. idiot distance:', cosdis(moronv, idiotv))
print('Moron vs. genius distance:', cosdis(moronv, geniusv))
print('Genius vs. prodigy distance:', cosdis(geniusv, prodigyv))

# And even opposite-meaning words tend to be closer than unrelated words
print('Moron vs. man distance:', cosdis(moronv, manv))
```

```
Moron vs. idiot distance: 0.5694909847846478
Moron vs. genius distance: 0.8327756328149172
Genius vs. prodigy distance: 0.6045973937007791
Moron vs. man distance: 0.9317691419289097
```

Properties of GloVe vectors

```
print('King-queen vs. man-woman:', cosdis(kingv - queenv, manv - womanv))

# Much more similar than gender-neutral "analogies" we could try to construct
print('\nKing-queen vs. man-person:', cosdis(kingv - queenv, manv - personv))

print('King-president vs. man-woman:', cosdis(kingv - presidentv, manv - womanv))

print('King-monarch vs. man-woman:', cosdis(kingv - monarchv, manv - womanv))

```

```
King-queen vs. man-woman: 0.40296734358842223

King-queen vs. man-person: 0.9490595508253746
King-president vs. man-woman: 0.9414983701409074
King-monarch vs. man-woman: 0.6771549397265793
```

Properties of GloVe vectors

```python
# It actually does pretty well on monarch, surprisingly.
print(f'Man vs. monarch:', cosdis(manv, monarchv))
print(f'Woman vs. monarch:', cosdis(womanv, monarchv))
# President favors men a bit, though less than I expected
print(f'\nMan vs. president:', cosdis(manv, presidentv))
print(f'Woman vs. president:', cosdis(womanv, presidentv))
# Doctor is pretty good!
print(f'\nMan vs. doctor:', cosdis(manv, doctorv))
print(f'Woman vs. doctor:', cosdis(womanv, doctorv))
# Nurse is still pretty gendered though.
print(f'\nMan vs. nurse:', cosdis(manv, nursev))
print(f'Woman vs. nurse:', cosdis(womanv, nursev))
# And CEO is too, though not as bad as nurse.
print(f'\nMan vs. CEO:', cosdis(manv, ceov))
print(f'Woman vs. CEO:', cosdis(womanv, ceov))
```

```
Man vs. monarch: 0.5922413733494826
Woman vs. monarch: 0.5941693025598238

Man vs. president: 0.5569893914832684
Woman vs. president: 0.6375253287060663

Man vs. doctor: 0.28804209610894094
Woman vs. doctor: 0.2747264697454299

Man vs. nurse: 0.428129645178737
Woman vs. nurse: 0.28449795808534417

Man vs. CEO: 0.7467859714356866
Woman vs. CEO: 0.8899819286150237
```

Reading and processing SST-2 dataset

```
1 display(dev_df)
```

	sentence	label	preprocessed
0	it 's a charming and often affecting journey .	1	it 's a charming and often affecting journey .
1	unflinchingly bleak and desperate	0	unflinchingly bleak and desperate
2	allows us to hope that nolan is poised to emba...	1	allows us to hope that nolan is poised to emba...
3	the acting , costumes , music , cinematography...	1	the acting , costumes , music , cinematography...
4	it 's slow -- very , very slow .	0	it 's slow -- very , very slow .
...	...	...	...
867	has all the depth of a wading pool .	0	has all the depth of a wading pool .
868	a movie with a real anarchic flair .	1	a movie with a real anarchic flair .
869	a subject like this should inspire reaction in...	0	a subject like this should inspire reaction in...
870	... is an arthritic attempt at directing by ca...	0	... is an arthritic attempt at directing by ca...
871	looking aristocratic , luminous yet careworn i...	1	looking aristocratic , luminous yet careworn i...

872 rows × 3 columns

Adding vectors for unknown and padding tokens

```python
glove_words = np.concatenate([glove_words, ['<unk>', '<pad>']])
glove_vocab['<unk>'] = len(glove_data)
glove_vocab['<pad>'] = len(glove_data)+1

unk_vector = np.mean(glove_vectors, axis=0)
pad_vector = np.zeros_like(unk_vector)
glove_vectors = np.concatenate([glove_vectors, [unk_vector, pad_vector]],axis=0)

print(glove_words)
print(glove_words.shape)
print(glove_vectors)
print(glove_vectors.shape)
```

```
['the' ',' '.' ... 'sandberger' '<unk>' '<pad>']
(400002,)
[[ 0.418       0.24968    -0.41242    ... -0.18411    -0.11514
  -0.78581   ]
 [ 0.013441    0.23682    -0.16899    ... -0.56657     0.044691
   0.30392   ]
 [ 0.15164     0.30177    -0.16763    ... -0.35652     0.016413
   0.10216   ]
 ...
 [ 0.072617   -0.51393     0.4728     ... -0.18907    -0.59021
   0.55559   ]
 [-0.12920061 -0.28866239 -0.01224894 ...  0.10069294  0.00653007
   0.0168515 ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]]
(400002, 50)
```

Adding vectors for unknown and padding tokens

```python
def preprocessed_to_ids(preprocessed):
  ids = []
  for word in preprocessed.split(' '): # We can count on being able to do this because we did the preprocessing above already
    if word in glove_vocab:
      ids.append(glove_vocab[word])
    else:
      ids.append(glove_vocab['<unk>'])
  return ids

train_df['input_ids'] = train_df['preprocessed'].apply(preprocessed_to_ids)
dev_df['input_ids'] = dev_df['preprocessed'].apply(preprocessed_to_ids)
display(dev_df)
```

	sentence	label	preprocessed	input_ids
0	it 's a charming and often affecting journey .	1	it 's a charming and often affecting journey .	[20, 9, 7, 12387, 5, 456, 7237, 3930, 2]
1	unflinchingly bleak and desperate	0	unflinchingly bleak and desperate	[101035, 12566, 5, 5317]
2	allows us to hope that nolan is poised to emba...	1	allows us to hope that nolan is poised to emba...	[2415, 95, 4, 824, 12, 13528, 14, 7490, 4, 174...
3	the acting , costumes , music , cinematography...	1	the acting , costumes , music , cinematography...	[0, 2050, 1, 10349, 1, 403, 1, 22181, 5, 1507,...
4	it 's slow -- very , very slow .	0	it 's slow -- very , very slow .	[20, 9, 2049, 65, 191, 1, 191, 2049, 2]
...	...	...	...	...
867	has all the depth of a wading pool .	0	has all the depth of a wading pool .	[31, 64, 0, 4735, 3, 7, 27989, 3216, 2]
868	a movie with a real anarchic flair .	1	a movie with a real anarchic flair .	[7, 1005, 17, 7, 567, 41588, 17056, 2]
869	a subject like this should inspire reaction in...	0	a subject like this should inspire reaction in...	[7, 1698, 117, 37, 189, 11356, 2614, 6, 47, 20...
870	... is an arthritic attempt at directing by ca...	0	... is an arthritic attempt at directing by ca...	[434, 14, 29, 57228, 1266, 22, 8044, 21, 63691...
871	looking aristocratic , luminous yet careworn i...	1	looking aristocratic , luminous yet careworn i...	[862, 21897, 1, 29085, 553, 203745, 6, 4917, 3...

872 rows × 4 columns

Dataset

```python
class SST2VectorDataset(Dataset):
  def __init__(self,
               labels=None,
               input_ids=None):

    self.y = torch.tensor(labels,dtype=torch.int64)
    self.input_ids = input_ids

  def __len__(self):
    return self.y.shape[0]

  def __getitem__(self, idx):
    rdict = {
      'y': self.y[idx],
      'input_ids': torch.tensor(self.input_ids[idx], dtype=torch.int64) # We generally want word IDs to be Longs
    }
    return rdict
```

```python
train_dataset = SST2VectorDataset(train_df['label'], train_df['input_ids'])
dev_dataset = SST2VectorDataset(dev_df['label'], dev_df['input_ids'])

print(train_dataset[0])
print(train_dataset[0]['input_ids'].shape)
```

```
{'y': tensor(0), 'input_ids': tensor([ 5708,    50, 52776,    25,     0, 13054,  1503])}
torch.Size([7])
```

DataLoader

```python
def SST2_collate(batch:List[Dict[str, torch.Tensor]]):
  y_batch = torch.tensor([example['y'] for example in batch])

  id_vectors = [example['input_ids'] for example in batch]

  # We're gonna pad these guys with the handy-dandy torch.nn.utils.rnn.pad_sequence
  # function, which takes a list of vectors and pads them out to the length of the
  # longest sequence in the list
  id_vector_matrix = torch.nn.utils.rnn.pad_sequence(id_vectors, batch_first=True, padding_value=glove_vocab['<pad>'])

  return {
      'y':y_batch,
      'input_ids':id_vector_matrix
  }
```

```python
batch_size = 10
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, collate_fn = SST2_collate, shuffle=True)
dev_dataloader = DataLoader(dev_dataset, batch_size=batch_size, collate_fn = SST2_collate, shuffle=False)
```

DataLoader

```python
torch.random.manual_seed(1234)
first_train_batch = next(iter(train_dataloader))
print('First training batch:')
print(first_train_batch)

print('First training batch sizes:')
print({key:value.shape for key, value in first_train_batch.items()})
```

```
First training batch:
{'y': tensor([0, 0, 0, 1, 1, 1, 1, 1, 0, 0]), 'input_ids': tensor([[   307,     66,      3,  11114,   2720,      5,   5097,  31351, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [ 42131, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [    29,  51710,  37369,   2692,     12,   1144,   1003,     64,    317,
           2516,      2, 400001, 400001, 400001, 400001, 400001],
        [  2322, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [ 18519, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [    32,   3478,     17,      1,      5,    907,     81,    757,     59,
            403,     81,    107,     36,     33,   1435,    106],
        [    12,  21590,    244,  21609, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [     4,   6636,   1121,   3954,     17,    319,  15215,      5,    608,
          33619,     17,   9693,   3861, 400001, 400001, 400001],
        [    14,     70,    151,      7,   1005, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001],
        [    20,    965,   1369,     70,     33,     81,  12681,     25,      0,
           2816,  88552,     20,      9,  16031, 400001, 400001]])}
First training batch sizes:
{'y': torch.Size([10]), 'input_ids': torch.Size([10, 16])}
```

Model

```python
class WordVectorLogisticRegression(pl.LightningModule):
  def __init__(self,
               word_vectors:np.ndarray,
               num_classes:int,
               learning_rate:float,
               padding_id:int,
               **kwargs):
    super().__init__( **kwargs)

    self.word_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(word_vectors),
                                                              freeze=True) #Typically we don't train the embedding layer
    self.output_layer  = torch.nn.Linear(word_vectors.shape[1], num_classes)
    self.learning_rate = learning_rate
    self.padding_id = padding_id # we'll need this later
    self.train_accuracy = Accuracy(task='binary')
    self.val_accuracy = Accuracy(task='binary')
```

Model

```python
def forward(self, y:torch.Tensor, input_ids:torch.Tensor):
    inputs_embeds = self.word_embeddings(input_ids) # shape (batch size, sequence length, embedding dim)
    padding_mask = (input_ids != self.padding_id).int()
    masked_sums = (padding_mask.unsqueeze(-1) * inputs_embeds).sum(dim=1)
    masked_counts = padding_mask.sum(dim=1)
    embedding_centroids = masked_sums/masked_counts.unsqueeze(-1)
    py_logits = self.output_layer(embedding_centroids.float())
    py = torch.argmax(py_logits, dim=1)
    loss = torch.nn.functional.cross_entropy(py_logits, y, reduction='mean')

    return {'py':py,
            'loss':loss}
```

Model

```
def configure_optimizers(self):
    return [torch.optim.Adam(self.parameters(), lr=self.learning_rate)]

def training_step(self, batch, batch_idx):
    result = self.forward(**batch)
    loss = result['loss']
    self.log('train_loss', result['loss'])
    self.train_accuracy.update(result['py'], batch['y'])
    return loss

def training_epoch_end(self, outs):
    print('Training accuracy:', self.train_accuracy.compute())

def validation_step(self, batch, batch_idx):
    result = self.forward(**batch)
    self.val_accuracy.update(result['py'], batch['y'])
    return result['loss']

def validation_epoch_end(self, outs):
    print('Validation accuracy:', self.val_accuracy.compute())
```

Model

```python
model = WordVectorLogisticRegression(word_vectors=glove_vectors,
                                     num_classes = 2,
                                     learning_rate = 0.01,
                                     padding_id = glove_vocab['<pad>'])

from pprint import pprint
with torch.no_grad():
  first_train_output = model(**first_train_batch)

print('First training output:')
pprint(first_train_output)

print('Output item shapes:')
pprint({key:value.shape for key, value in first_train_output.items()})
```

```
First training output:
{'loss': tensor(0.7152), 'py': tensor([1, 0, 1, 1, 0, 1, 1, 1, 1, 1])}
Output item shapes:
{'loss': torch.Size([]), 'py': torch.Size([10])}
```

Trainer

```python
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks.progress import TQDMProgressBar

trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=3,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.5,
    )
```

Trainer

```
trainer.fit(model=model,
            train_dataloaders=train_dataloader,
            val_dataloaders=dev_dataloader)
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name            | Type           | Params
---------------------------------------------------
0 | word_embeddings | Embedding      | 20.0 M
1 | output_layer    | Linear         | 102
2 | train_accuracy  | BinaryAccuracy | 0
3 | val_accuracy    | BinaryAccuracy | 0
---------------------------------------------------
102       Trainable params
20.0 M    Non-trainable params
20.0 M    Total params
80.001    Total estimated model params size (MB)
Validation accuracy: tensor(0.5000, device='cuda:0')
```

Epoch 2: 100% 6911/6911 [00:31<00:00, 220.96it/s, loss=0.562, v_num=6]

```
Validation accuracy: tensor(0.7197, device='cuda:0')
Validation accuracy: tensor(0.7188, device='cuda:0')
Training accuracy: tensor(0.7490, device='cuda:0')
Validation accuracy: tensor(0.7109, device='cuda:0')
Validation accuracy: tensor(0.7161, device='cuda:0')
Training accuracy: tensor(0.7501, device='cuda:0')
Validation accuracy: tensor(0.7142, device='cuda:0')
Validation accuracy: tensor(0.7155, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=3` reached.
Training accuracy: tensor(0.7502, device='cuda:0')
```

Concluding thoughts

Word vector models

- Word2Vec
  - CBOW
  - Skip-gram
- GloVe

Word vectors in classification

- Padding
- Collation
- Centroids