Natural Language Inference: Modeling strategies

Christopher Potts

Stanford Linguistics

CS224u: Natural language understanding

Hand-built features

Hand-built feature ideas

Hand-built feature ideas

1. Word overlap

Hand-built feature ideas

1. Word overlap
2. Word cross-product

Hand-built feature ideas

1. Word overlap

2. Word cross-product

3. Additional WordNet relations

Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance

Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)

Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features

Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features
7. Negation

Hand-built feature ideas

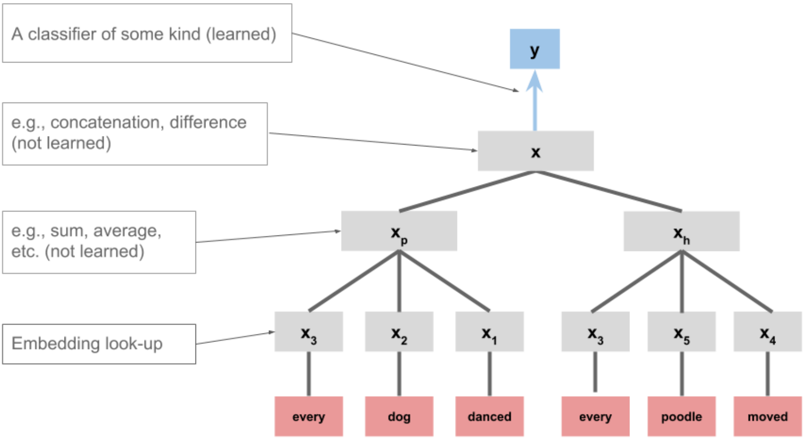
1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features
7. Negation
8. Quantifier relations (e.g., *every* $\sqsubset$ *some*; see MacCartney and Manning 2009)

Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features
7. Negation
8. Quantifier relations (e.g., *every* $\sqsubset$ *some*; see MacCartney and Manning 2009)
9. Named entity features

Sentence-encoding models

Distributed representations as features



Code: Distributed representations as features

```
[1]: import numpy as np
     import os
     from sklearn.linear_model import LogisticRegression
     import nli, utils
```

```
[2]: SNLI_HOME = os.path.join("data", "nlidata", "snli_1.0")
     GLOVE_HOME = os.path.join('data', 'glove.6B')
```

```
[3]: glove_lookup = utils.glove2dict(
         os.path.join(GLOVE_HOME, 'glove.6B.50d.txt'))
```

```
[4]: def _get_tree_vecs(tree, lookup, np_func):
         allvecs = np.array([lookup[w] for w in tree.leaves() if w in lookup])
         if len(allvecs) == 0:
             dim = len(next(iter(lookup.values())))
             feats = np.zeros(dim)
         else:
             feats = np_func(allvecs, axis=0)
         return feats
```

```
[5]: def glove_leaves_phi(t1, t2, np_func=np.sum):
         prem_vecs = _get_tree_vecs(t1, glove_lookup, np_func)
         hyp_vecs = _get_tree_vecs(t2, glove_lookup, np_func)
         return np.concatenate((prem_vecs, hyp_vecs))
```

```
[6]: def glove_leaves_sum_phi(t1, t2):
         return glove_leaves_phi(t1, t2, np_func=np.sum)
```

Code: Distributed representations as features

[7]:
```
def fit_softmax(X, y):
    mod = LogisticRegression(
        fit_intercept=True, solver='liblinear', multi_class='auto')
    mod.fit(X, y)
    return mod
```

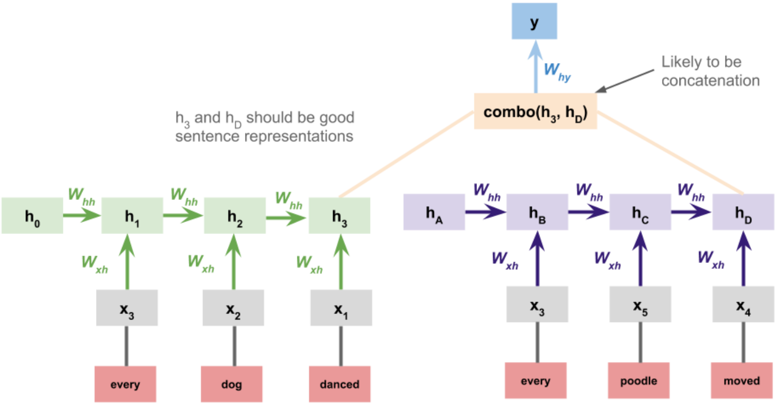
[8]:
```
glove_sum_experiment = nli.experiment(
    nli.SNLITrainReader(SNLI_HOME),
    glove_leaves_sum_phi,
    fit_softmax,
    assess_reader=nli.SNLIDevReader(SNLI_HOME),
    vectorize=False) # We already have vectors!
```

Rationale for sentence-encoding models

1. Encoding the premise and hypothesis separately might give the model a chance to find rich abstract relationships between them.

2. Sentence-level encoding could facilitate transfer to other tasks (Dagan et al.’s (2006) vision).

Sentence-encoding RNNs



PyTorch strategy: Sentence-encoding RNNs

The full implementation is in `nli_02_models.ipynb`.

`TorchRNNSentenceEncoderDataset`

This is conceptually a list of pairs of sequences, each with their lengths, and a label vector:

$$\left[\left([\texttt{every}, \texttt{dog}, \texttt{danced}], [\texttt{every}, \texttt{poodle}, \texttt{moved}]\right), (3, 3), \textbf{entailment}\right]$$

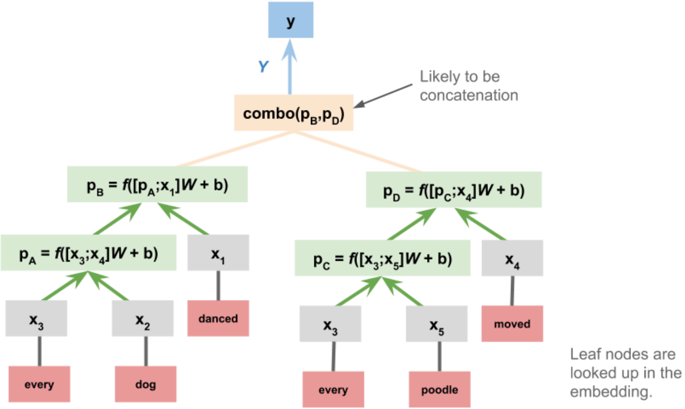
`TorchRNNSentenceEncoderClassifierModel`

This is concetually a premise RNN and a hypothesis RNN. The `forward` method uses them to process the two parts of the example, concatenate the outputs of those passes, and feed them into a classifier.

`TorchRNNSentenceEncoderClassifier`

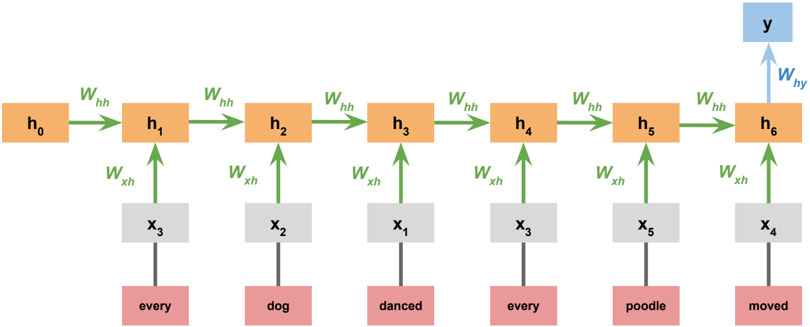
This is basically unchanged from its super class `TorchRNNClassifier`, except the `predict_proba` method needs to deal with the new example format.

Sentence-encoding TreeNNs



Chained models

Simple RNN


y
W hy
W hh
h0
h1
h2
h3
h4
h5
h6
W xh
x3
x2
x1
x3
x5
x4
every
dog
danced
every
poodle
moved

Rationale for chained models

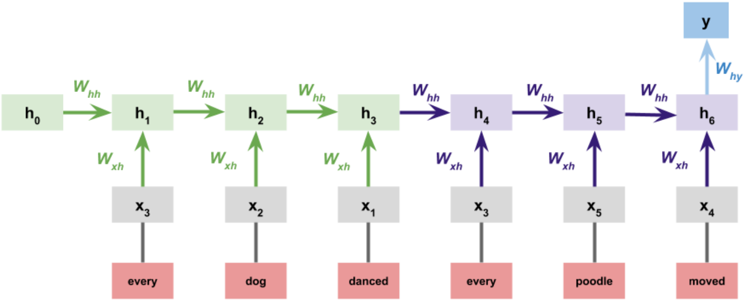
1. The premise truly establishes the context for the hypothesis.

2. Might be seen as corresponding to a real processing model.

Code snippet: Simple RNN

```
[1]: import os
     from torch_rnn_classifier import TorchRNNClassifier
     import nli, utils

[2]: SNLI_HOME = os.path.join("data", "nlidata", "snli_1.0")

[3]: def simple_chained_rep_rnn_phi(t1, t2):
         return t1.leaves() + ["[SEP]"] + t2.leaves()

[4]: def fit_simple_chained_rnn(X, y):
         vocab = utils.get_vocab(X, n_words=10000)
         vocab.append("[SEP]")
         mod = TorchRNNClassifier(vocab, hidden_dim=50, max_iter=50)
         mod.fit(X, y)
         return mod

[5]: simple_chained_rnn_experiment = nli.experiment(
         nli.SNLITrainReader(SNLI_HOME, samp_percentage=0.10),
         simple_chained_rep_rnn_phi,
         fit_simple_chained_rnn,
         vectorize=False)
```

Premise and hypothesis RNNs



The PyTorch implementation strategy is similar to the one outlined earlier for sentence-encoding RNNs, except the final hidden state of the premise RNN becomes the initial hidden state for the hypothesis RNN.

Other strategies

`TorchRNNClassifier`

- `TorchRNNClassifier` feeds its final hidden state directly to the classifier layer.
- If `bidirectional=True`, then the two final states are concatenated and fed directly to the classifier layer.

Other ideas

- *Pool* all the hidden states with **max** or **mean**.
- Different pooling options can be combined.
- Additional layers between the hidden representation (however defined) and the classifier layer.
- Attention mechanisms

References I

Ido Dagan, Oren Glickman, and Bernardo Magnini. 2006. The PASCAL recognising textual entailment challenge. In *Machine Learning Challenges, Lecture Notes in Computer Science*, volume 3944, pages 177–190. Springer-Verlag.

Bill MacCartney and Christopher D. Manning. 2009. An extended model of natural logic. In *Proceedings of the Eighth International Conference on Computational Semantics*, pages 140–156, Tilburg, The Netherlands. Association for Computational Linguistics.