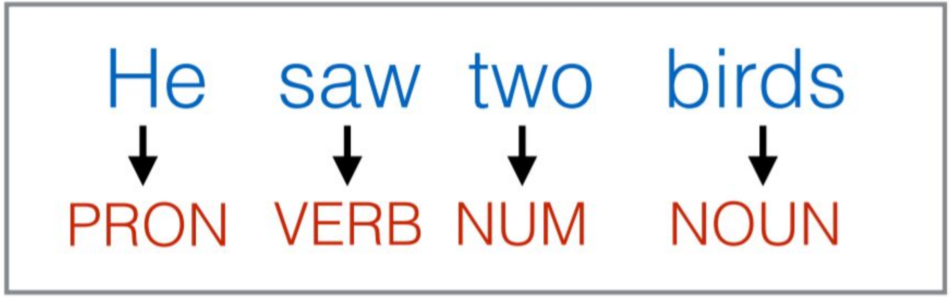
Assignment 1: Multilingual POS tagging

Recitation: Sept 14, 2020

Sachin Kumar

Parts of Speech

- Lexical Categories or Word Classes or Tags



Open vs Closed Class Words

- Closed
  - Determiners: a an the that
  - Prepositions: at from
  - Pronouns
- Open
  - Nouns, Verbs, (Adjectives, Adverbs)

POS tagging

- A word can has multiple potential POS tags
  - The back door
  - On my back
  - Win the voters back
  - Promised to back the bill
- Open class and unseen word, eg TikTok
  - Noun, Verbs, (or even adjective, adverb)

- Have to determine the POS tag of a particular instance of the word
- Formulate as a learning problem
  - Needs supervision: training data with text and marked POS tags

Sources of Information to determine POS tag

- Neighboring POS tags
  - Bill saw that man yesterday

- Knowledge of word probabilities
  - Man is rarely used as a verb

Latter proves the most useful but former also helps

Simple Example: Feature based tagger

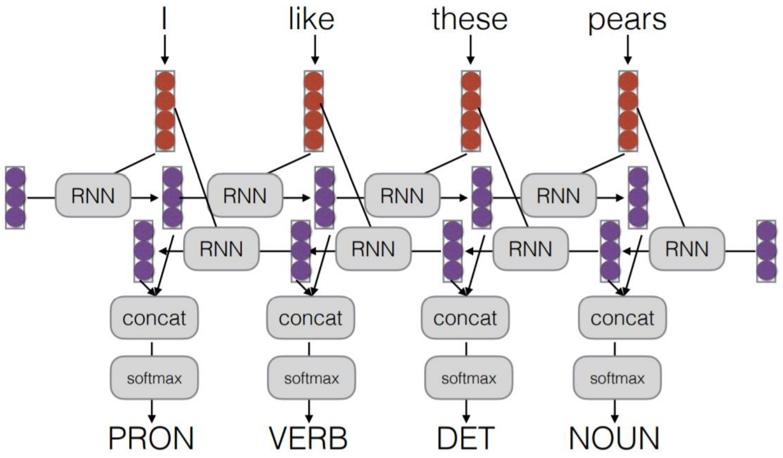
- Can do surprisingly well by looking at the word itself

Word	the: the → DT
Lowercased word	Importantly: importantly → RB
Prefixes	unfathomable: un- → JJ
Suffixes	Importantly: -ly → RB
Capitalization	Meridian: CAP → NNP
Word shapes	35-year: d-x → JJ

- Learn a maxent model p(t|w)

Baseline Model

- Use a Bidirectional LSTM to generate features of every token
  - Features are contextual on surrounding text (but not the tags)



Multilingual POS tagging

- POS tagging on multiple languages
- Different languages usually define different sets of POS tags
    - Universal dependencies: Unify the POS tags across languages
- Amount of labeled data varies across languages
    - Low resource languages might benefit from high resource ones.

Requirements

- Machine with a GPU
  - AWS
  - Or , your own computer

- Software/packages/libraries:
  - Conda (recommended)
  - Python >=3.6
  - Pytorch>=1.0
  - Torchtext 0.7

Code Organization

- assign1/
    - config.json
    - model.py
    - main.py
    - saved_models/
        - en-model.pt
    - data/
        - en/ es/ af/ cs/ ar/ lt/ hy/ ta/
            - train dev test

```json
{
    "embedding_dim": 100,
    "hidden_dim":128,
    "n_layers":2,
    "bidirectional":true,
    "dropout":0.25,
    "batch_size": 128
}
```

Model definition: model.py

```python
import torch
import torch.nn as nn


class BiLSTMPOSTagger(nn.Module):
    def __init__(...
    ): ...

    def forward(self, text): ...
```

```python
self.embedding = nn.Embedding(input_dim, embedding_dim, padding_idx=pad_idx)

self.lstm = nn.LSTM(
    embedding_dim,
    hidden_dim,
    num_layers=n_layers,
    bidirectional=bidirectional,
    dropout=dropout if n_layers > 1 else 0,
)

self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

self.dropout = nn.Dropout(dropout)
```

```python
embedded = self.dropout(self.embedding(text))
outputs, (hidden, cell) = self.lstm(embedded)
predictions = self.fc(self.dropout(outputs))

return predictions
```

Loading the data: main.py

```
TEXT = data.Field(lower=True)
UD_TAGS = data.Field()

fields = (("text", TEXT), ("udtags", UD_TAGS))

# load the data from the specific path
train_data, valid_data, test_data = datasets.UDPOS.splits(
    fields=fields,
    path=os.path.join("data", args.lang),
    train="{}-ud-train.conll".format(args.lang),
    validation="{}-ud-dev.conll".format(args.lang),
    test="{}-ud-test.conll".format(args.lang),
)  # modify this to include our own dataset
# building the vocabulary for both text and the labels
MIN_FREQ = 2

TEXT.build_vocab(train_data, min_freq=MIN_FREQ)
UD_TAGS.build_vocab(train_data)
```

Creating the model, loss and optimizer

```
model = BiLSTMPOSTagger(
    input_dim=len(TEXT.vocab),
    embedding_dim=params["embedding_dim"],
    hidden_dim=params["hidden_dim"],
    output_dim=len(UD_TAGS.vocab),
    n_layers=params["n_layers"],
    bidirectional=params["bidirectional"],
    dropout=params["dropout"],
    pad_idx=PAD_IDX,
)
```

```
criterion = nn.CrossEntropyLoss(ignore_index=TAG_PAD_IDX)
```

```
optimizer = optim.Adam(model.parameters())
```

Train and evaluate the model

```
model.train()

for batch in iterator:
    text = batch.text
    tags = batch.udtags

    optimizer.zero_grad()
    predictions = model(text)

    loss = criterion(predictions, tags)
    loss.backward()
    optimizer.step()
```

Grading

- Train and reproduce the results on the given datasets
  - B+
- Report with analysis: A-
  - Performance across language family, typology, datasize…
  - Hyperparameter tuning: config.json
  - Performance across tag types
- Non-trivial extension which leads to improvement in scores: A, A+

Extension: Initialize with pretrained embeddings

- Load the vectors from file: main.py

```
TEXT.build_vocab(train_data,
                 min_freq = MIN_FREQ,
                 vectors = "glove.6B.300d",
                 unk_init = torch.Tensor.normal_)
```

- Initialize embedding table with the pretrained vectors: main.py

```
pretrained_embeddings = TEXT.vocab.vectors
model.embedding.weight.data.copy_(pretrained_embeddings)
```

- Can either fix or train the embeddings

Extension: Initialize with a language model

1. Train the LSTMs with a language model objective (like ELMo [1]) using the same data.
    a. Initialize the embedding AND the LSTMs with this pretrained LM parameters. Randomly initialize the final layer and fine-tune with POS tagging loss.

2. Other: Initialize with BERT-like models [2]
    a. Might have to figure out tokenization.

[1] Deep contextualized word representations. Peters et al. NAACL 2018
[2] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Devlin et al. NAACL 2019

Extensions: Character Embedding

- Modify Field “TEXT” to preprocess characters as well as words
    - data.Field → [data.NestedField]

- Modify embedding in model.py to accept characters.

```
self.embedding = nn.Embedding(input_dim, embedding_dim, padding_idx=pad_idx)
```

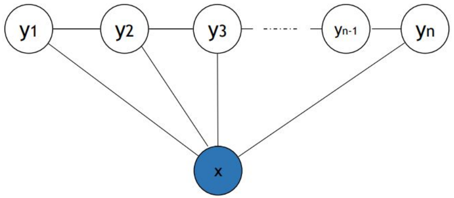
↓

```
self.conv1 = nn.Sequential(
    nn.Conv1d(args.num_features, 256, kernel_size=7, stride=1),
    nn.ReLU(),
    nn.MaxPool1d(kernel_size=3, stride=3)
)
```

Extension: Structure of labels (CRFs)

- Till now, we only used the given text to predict the labels.
- Here: use the other labels to influence the current labels: BiLSTM-CRFs

First-order linear CRF



$$P(Y|X) = \frac{\prod_{i=1}^{L} \psi_i(y_{i-1}, y_i, X)}{\sum_{Y'} \prod_{i=1}^{L} \psi_i(y'_{i-1}, y'_i, X)}$$

- Training: forward-backward algorithm. Decoding: Viterbi algorithm
- Example: https://pytorch-crf.readthedocs.io/en/stable/

Other Extensions

- Other Losses: https://arxiv.org/abs/1604.05529
- Adversarial Training: https://www.aclweb.org/anthology/N18-1089/
- Multi-task learning: https://arxiv.org/ftp/arxiv/papers/1807/1807.00818.pdf