Natural Language Processing
Tutorial 2
Neural dependency parsing

Francesco Cazzaro

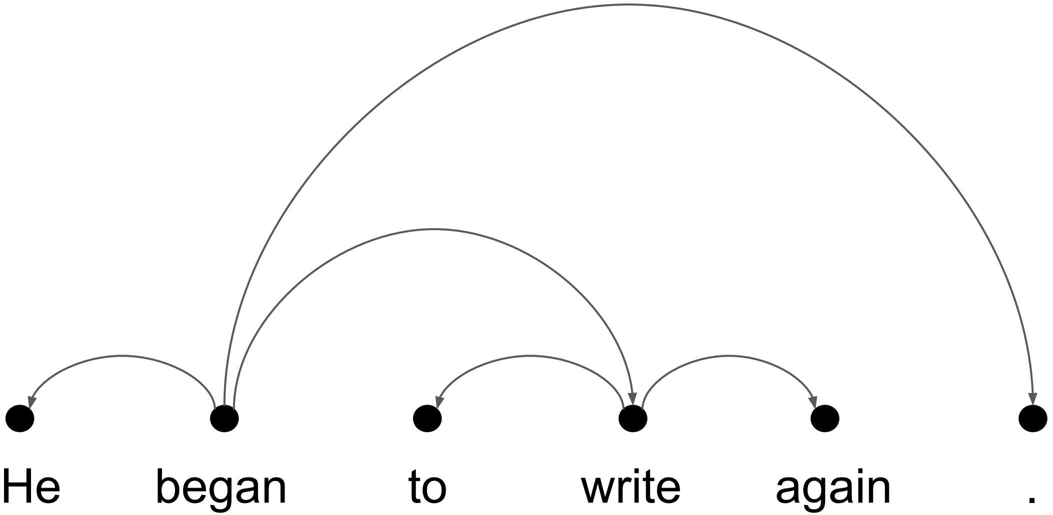
Notebook Goal: From theory to practice

Starting from scratch:

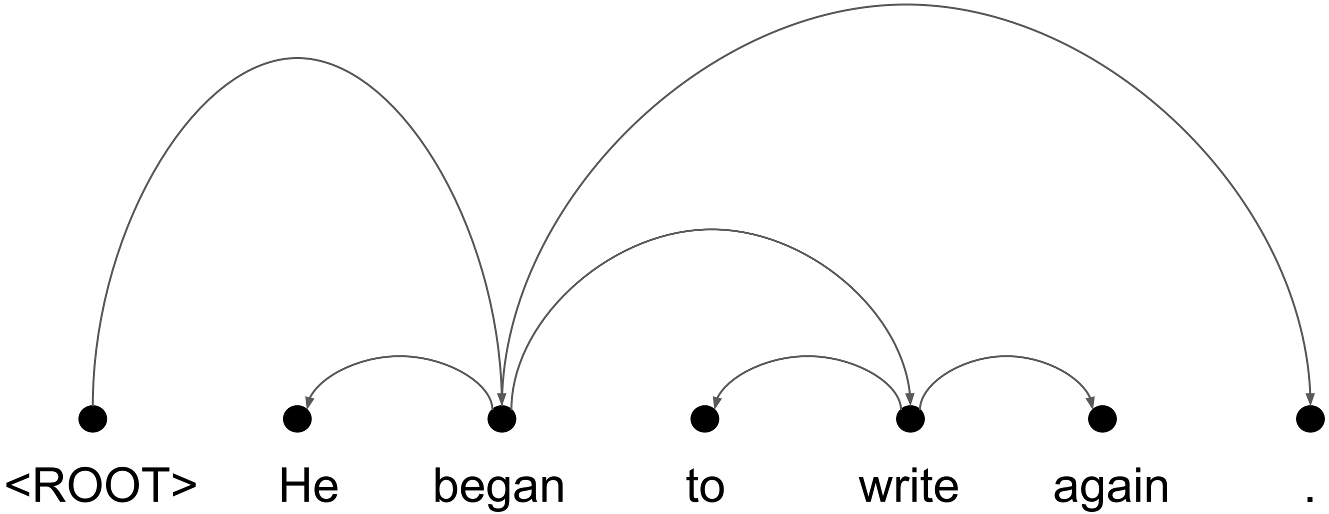
- Implement the Arc-standard parser
- Implement an Oracle
- Train a neural model

Reference paper:
Kiperwasser and Goldberg, Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations
Transactions of the Association for Computational Linguistics, Volume 4, 2016.

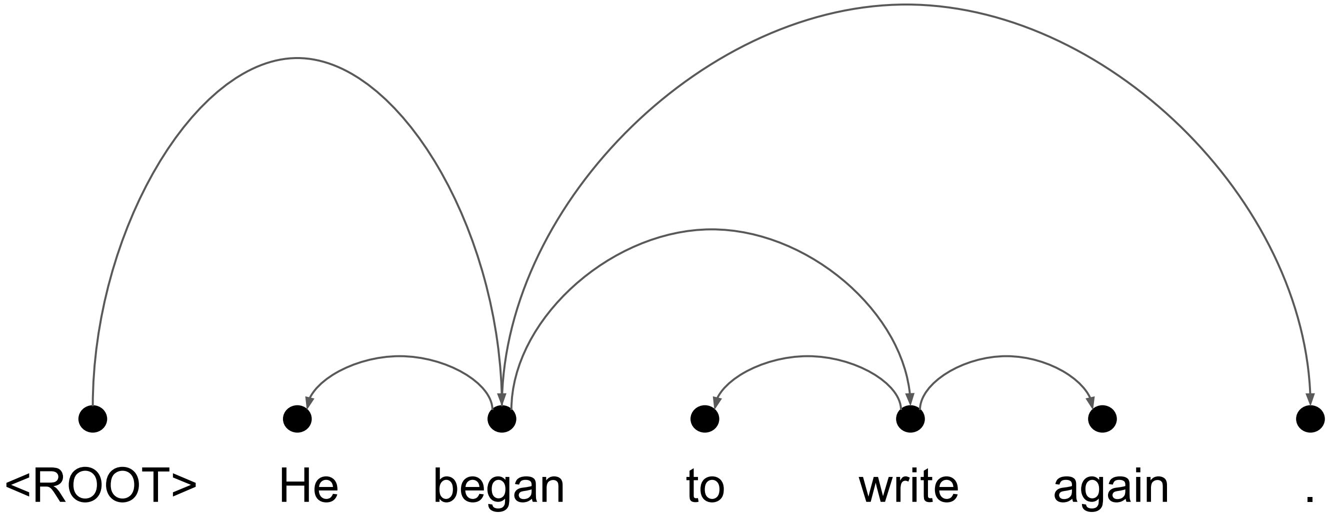
Representing the tree in python



Representing the tree in python



Representing the tree in python

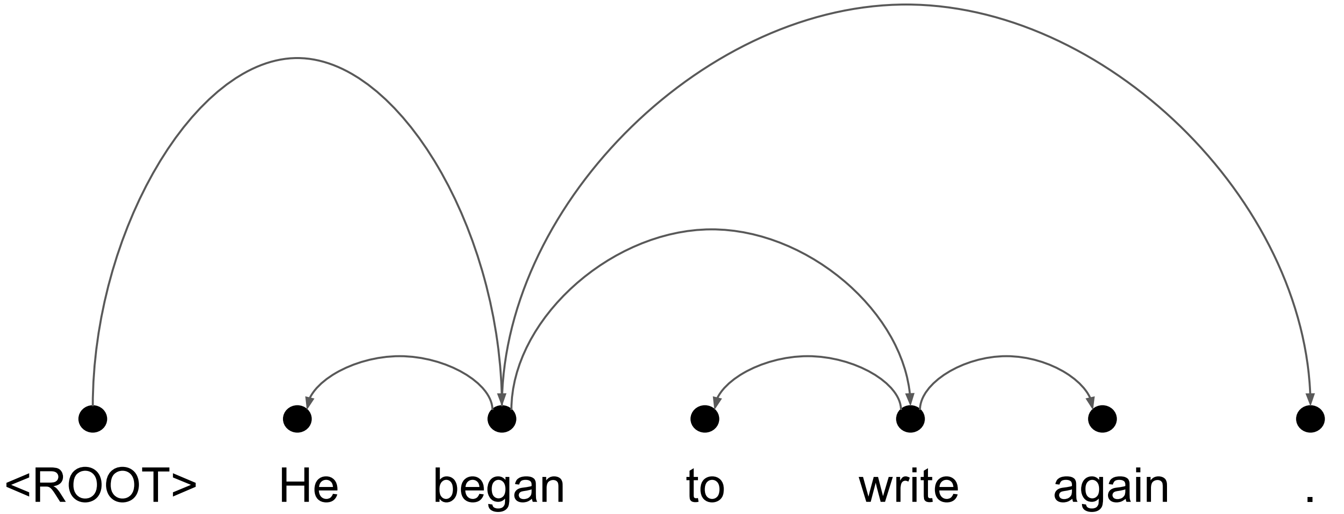


<ROOT> He began to write again .

Array

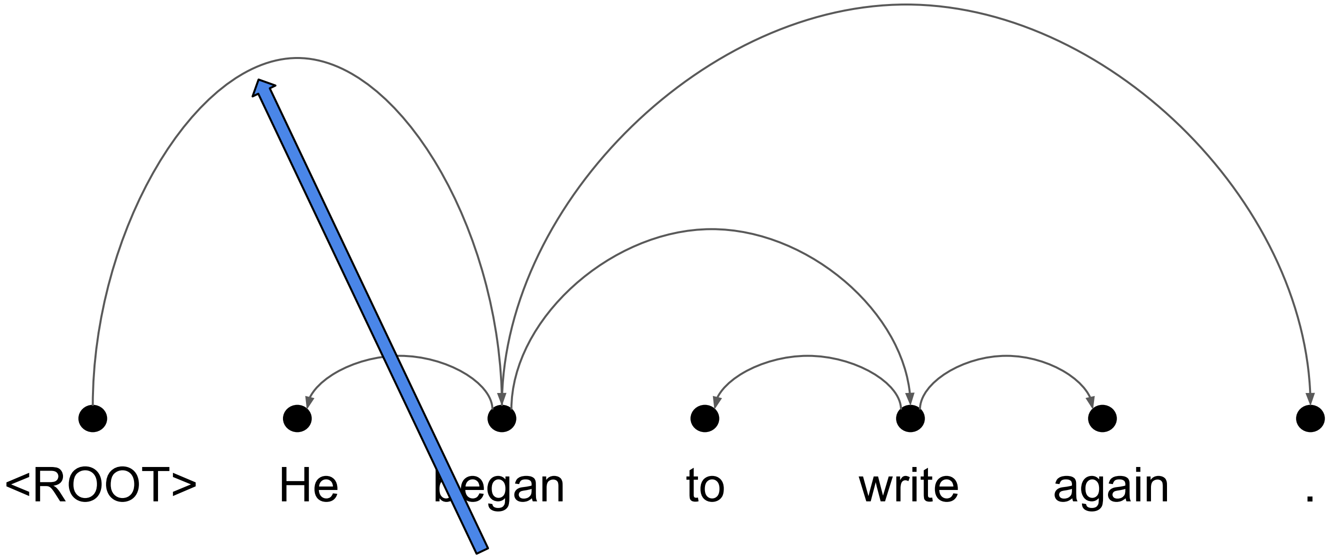
[ ]

Representing the tree in python



[-1, 2, 0, 4, 2, 4, 2]

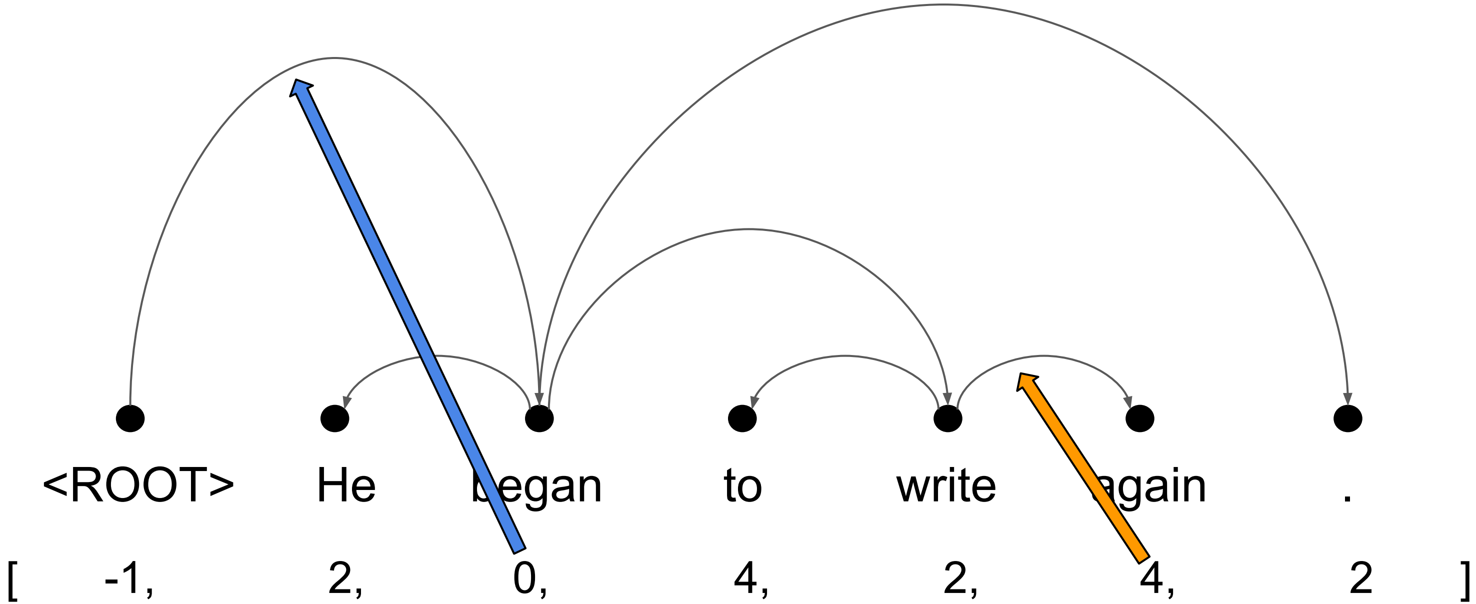
Representing the tree in python



<ROOT> He began to write again .

[-1, 2, 0, 4, 2, 4, 2]

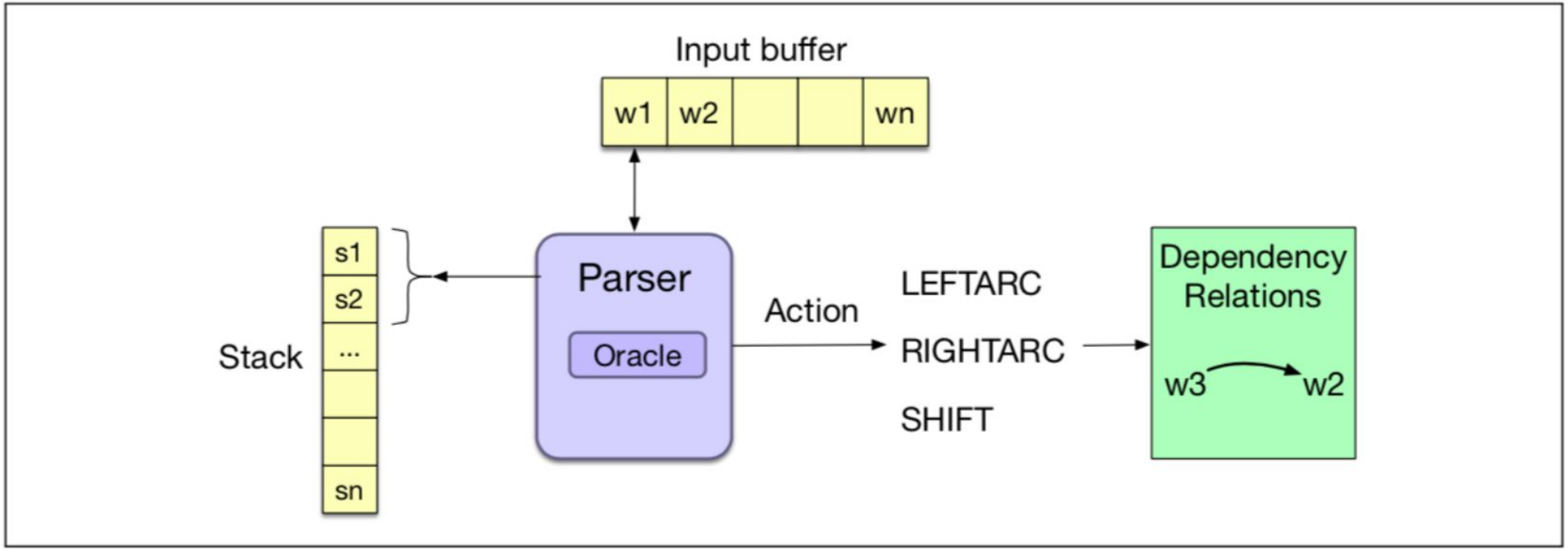
Representing the tree in python



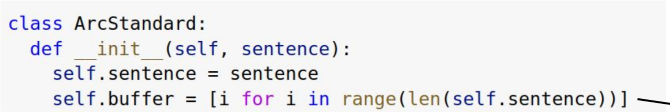
<ROOT> He began to write again .

[-1, 2, 0, 4, 2, 4, 2]

Arc-Standard Parser



Arc-Standard Parser

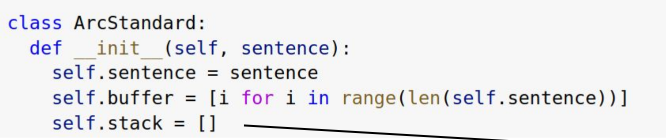


```
class ArcStandard:
  def __init__(self, sentence):
    self.sentence = sentence
    self.buffer = [i for i in range(len(self.sentence))]
```

b: [<ROOT>, He, began, to, write, again, .]

b: [0, 1, 2, 3, 4, 5, 6]

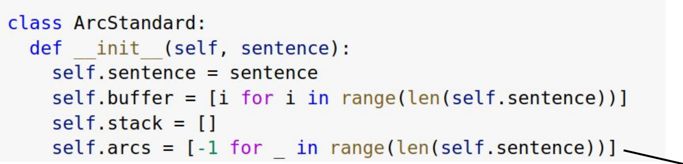
Arc-Standard Parser



```
class ArcStandard:
  def __init__(self, sentence):
    self.sentence = sentence
    self.buffer = [i for i in range(len(self.sentence))]
    self.stack = []
```

b: [<ROOT>, He, began, to, write, again, .]
s: [ ]
b: [0, 1, 2, 3, 4, 5, 6]
s: [ ]

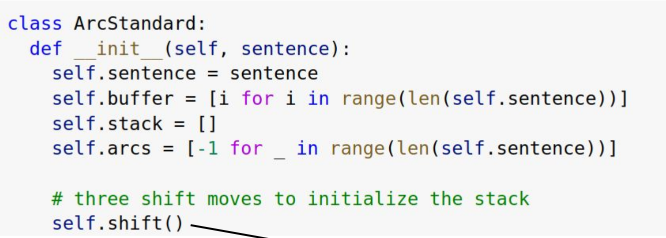
Arc-Standard Parser



```
class ArcStandard:
  def __init__(self, sentence):
    self.sentence = sentence
    self.buffer = [i for i in range(len(self.sentence))]
    self.stack = []
    self.arcs = [-1 for _ in range(len(self.sentence))]
```

b: [<ROOT>, He, began, to, write, again, .]
s: [ ]
b: [0, 1, 2, 3, 4, 5, 6]
s: [ ]
a: [-1, -1, -1, -1, -1, -1, -1]

Arc-Standard Parser

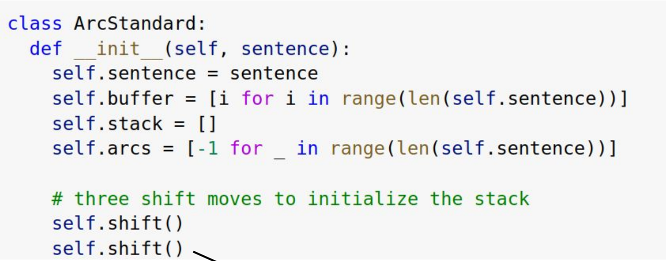


```
class ArcStandard:
  def __init__(self, sentence):
    self.sentence = sentence
    self.buffer = [i for i in range(len(self.sentence))]
    self.stack = []
    self.arcs = [-1 for _ in range(len(self.sentence))]

    # three shift moves to initialize the stack
    self.shift()
```

b: [<ROOT>, He, began, to, write, again, .]
s: [ ]
b: [0, 1, 2, 3, 4, 5, 6]
s: [ ]
a: [-1, -1, -1, -1, -1, -1, -1]

b: [He, began, to, write, again, .]
s: [<ROOT>]
b: [1, 2, 3, 4, 5, 6]
s: [0]

Arc-Standard Parser



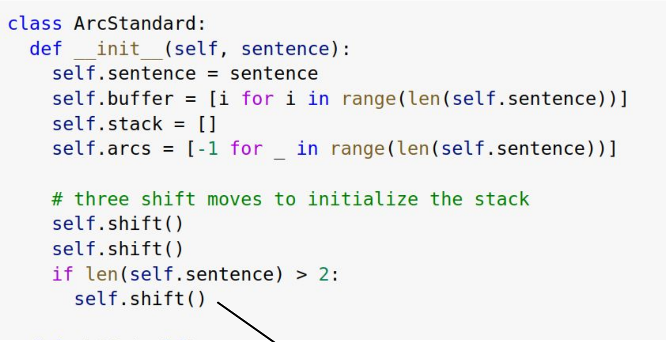
```
class ArcStandard:
  def __init__(self, sentence):
    self.sentence = sentence
    self.buffer = [i for i in range(len(self.sentence))]
    self.stack = []
    self.arcs = [-1 for _ in range(len(self.sentence))]

    # three shift moves to initialize the stack
    self.shift()
    self.shift()
```

b: [<ROOT>, He, began, to, write, again, .]
s: [ ]
b: [0, 1, 2, 3, 4, 5, 6]
s: [ ]
a: [-1, -1, -1, -1, -1, -1, -1]

b: [He, began, to, write, again, .]
s: [<ROOT>]
b: [1, 2, 3, 4, 5, 6]
s: [0]

b: [began, to, write, again, .]
s: [<ROOT>, He]
b: [2, 3, 4, 5, 6]
s: [0, 1]

Arc-Standard Parser



```python
class ArcStandard:
  def __init__(self, sentence):
    self.sentence = sentence
    self.buffer = [i for i in range(len(self.sentence))]
    self.stack = []
    self.arcs = [-1 for _ in range(len(self.sentence))]

    # three shift moves to initialize the stack
    self.shift()
    self.shift()
    if len(self.sentence) > 2:
      self.shift()
```

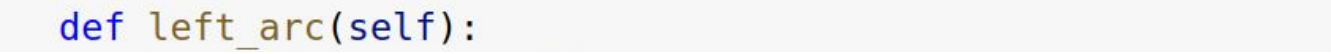
b: [<ROOT>, He, began, to, write, again, .]
s: [ ]
b: [0, 1, 2, 3, 4, 5, 6]
s: [ ]
a: [-1, -1, -1, -1, -1, -1, -1]

b: [He, began, to, write, again, .]
s: [<ROOT>]
b: [1, 2, 3, 4, 5, 6]
s: [0]

b: [began, to, write, again, .]
s: [<ROOT>, He]
b: [2, 3, 4, 5, 6]
s: [0, 1]

b: [to, write, again, .]
s: [<ROOT>, He, began]
b: [3, 4, 5, 6]
s: [0, 1, 2]

Left-Arc



```
def left_arc(self):
```

b: [to, write, again, .]
s: [<ROOT>, He, began]
b: [3, 4, 5, 6]
s: [0, 1, 2]
a: [-1, -1, -1, -1, -1, -1, -1]

Goal

b: [to, write, again, .]
s: [<ROOT>, began]
b: [3, 4, 5, 6]
s: [0, 2]
a: [-1, 2, -1, -1, -1, -1, -1]

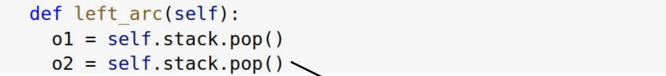
Left-Arc

```
def left_arc(self):
  o1 = self.stack.pop()
```

b: [to, write, again, .]
s: [<ROOT>, He, began]
b: [3, 4, 5, 6]
s: [0, 1, 2]
a: [-1, -1, -1, -1, -1, -1, -1]

b: [to, write, again, .]
s: [<ROOT>, He], o1 = began
b: [3, 4, 5, 6]
s: [0, 1], o1 = 2

Left-Arc



```
def left_arc(self):
  o1 = self.stack.pop()
  o2 = self.stack.pop()
```

b: [to, write, again, .]
s: [<ROOT>, He, began]
b: [3, 4, 5, 6]
s: [0, 1, 2]
a: [-1, -1, -1, -1, -1, -1, -1]

b: [to, write, again, .]
s: [<ROOT>, He], o1 = began
b: [3, 4, 5, 6]
s: [0, 1], o1 = 2

b: [to, write, again, .]
s: [<ROOT>], o1 = began, o2 = He
b: [3, 4, 5, 6]
s: [0], o1 = 2, o2 = 1

Left-Arc

```
def left_arc(self):
  o1 = self.stack.pop()
  o2 = self.stack.pop()
  self.arcs[o2] = o1
```

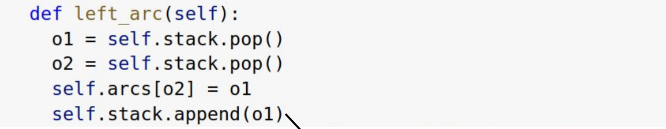
b: [to, write, again, .]
s: [<ROOT>, He, began]
b: [3, 4, 5, 6]
s: [0, 1, 2]
a: [-1, -1, -1, -1, -1, -1, -1]

b: [to, write, again, .]
s: [<ROOT>, He], o1 = began
b: [3, 4, 5, 6]
s: [0, 1], o1 = 2

b: [to, write, again, .]
s: [<ROOT>], o1 = began, o2 = He
b: [3, 4, 5, 6]
s: [0], o1 = 2, o2 = 1

o1 = 2, o2 = 1
a: [-1, 2, -1, -1, -1, -1, -1]

Left-Arc



```
def left_arc(self):
  o1 = self.stack.pop()
  o2 = self.stack.pop()
  self.arcs[o2] = o1
  self.stack.append(o1)
```

b: [to, write, again, .]
s: [<ROOT>, He, began]
b: [3, 4, 5, 6]
s: [0, 1, 2]
a: [-1, -1, -1, -1, -1, -1, -1]

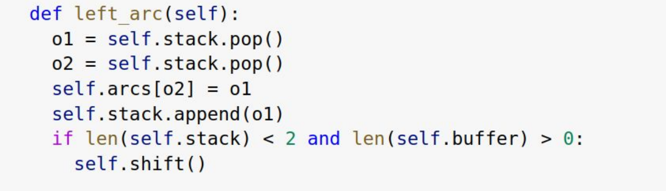
b: [to, write, again, .]
s: [<ROOT>, He], o1 = began
b: [3, 4, 5, 6]
s: [0, 1], o1 = 2

b: [to, write, again, .]
s: [<ROOT>], o1 = began, o2 = He
b: [3, 4, 5, 6]
s: [0], o1 = 2, o2 = 1

o1 = 2, o2 = 1
a: [-1, 2, -1, -1, -1, -1, -1]

b: [to, write, again, .]
s: [<ROOT>, began]
b: [3, 4, 5, 6]
s: [0, 2]
a: [-1, 2, -1, -1, -1, -1, -1]

Left-Arc



```
def left_arc(self):
  o1 = self.stack.pop()
  o2 = self.stack.pop()
  self.arcs[o2] = o1
  self.stack.append(o1)
  if len(self.stack) < 2 and len(self.buffer) > 0:
    self.shift()
```

b: [to, write, again, .]
s: [<ROOT>, He, began]
b: [3, 4, 5, 6]
s: [0, 1, 2]
a: [-1, -1, -1, -1, -1, -1, -1]

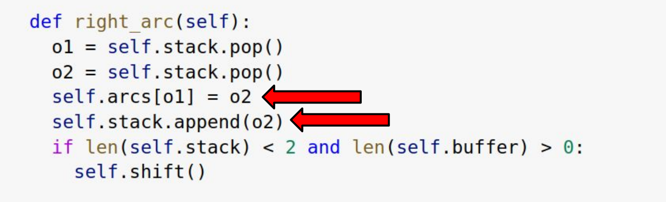
b: [to, write, again, .]
s: [<ROOT>, He], o1 = began
b: [3, 4, 5, 6]
s: [0, 1], o1 = 2

b: [to, write, again, .]
s: [<ROOT>], o1 = began, o2 = He
b: [3, 4, 5, 6]
s: [0], o1 = 2, o2 = 1

o1 = 2, o2 = 1
a: [-1, 2, -1, -1, -1, -1, -1]

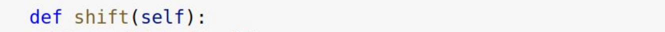
b: [to, write, again, .]
s: [<ROOT>, began]
b: [3, 4, 5, 6]
s: [0, 2]
a: [-1, 2, -1, -1, -1, -1, -1]

Right-Arc



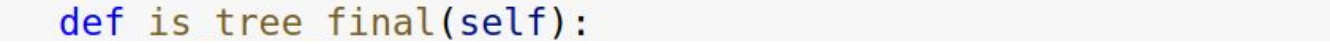
```python
def right_arc(self):
  o1 = self.stack.pop()
  o2 = self.stack.pop()
  self.arcs[o1] = o2
  self.stack.append(o2)
  if len(self.stack) < 2 and len(self.buffer) > 0:
    self.shift()
```

Your turn!



```
def shift(self):
```

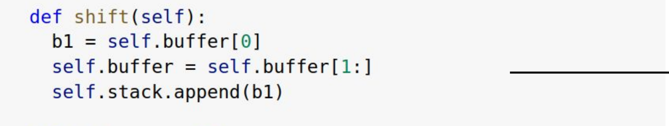
?



```
def is tree final(self):
```

?

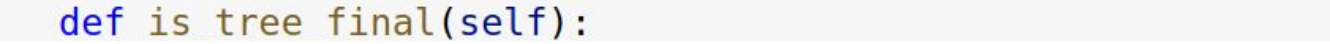
Solution



```
def shift(self):
  b1 = self.buffer[0]
  self.buffer = self.buffer[1:]
  self.stack.append(b1)
```

b: [to, write, again, .]
s: [<ROOT>, began]
b: [3, 4, 5, 6]
s: [0, 2]
a: [-1, 2, -1, -1, -1, -1, -1]

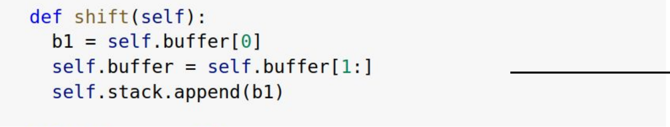
b: [write, again, .]
s: [<ROOT>, began, to]
b: [4, 5, 6]
s: [0, 2, 3]
a: [-1, 2, -1, -1, -1, -1, -1]



```
def is tree final(self):
```

?

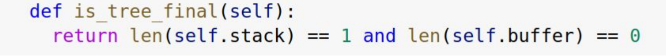
Solution



```python
def shift(self):
  b1 = self.buffer[0]
  self.buffer = self.buffer[1:]
  self.stack.append(b1)
```

b: [to, write, again, .]
s: [<ROOT>, began]
b: [3, 4, 5, 6]
s: [0, 2]
a: [-1, 2, -1, -1, -1, -1, -1]

b: [write, again, .]
s: [<ROOT>, began, to]
b: [4, 5, 6]
s: [0, 2, 3]
a: [-1, 2, -1, -1, -1, -1, -1]



```python
def is_tree_final(self):
  return len(self.stack) == 1 and len(self.buffer) == 0
```

Arc-Standard Parser

```
sentence = ["<ROOT>", "He", "began", "to", "write", "again", "."]
gold = [-1, 2, 0, 4, 2, 4, 2 ]

parser = ArcStandard(sentence)
parser.print_configuration()
```

```
['<ROOT>', 'He', 'began'] ['to', 'write', 'again', '.']
[-1, -1, -1, -1, -1, -1, -1]
```

```
parser.left_arc()
parser.print_configuration()
```

```
['<ROOT>', 'began'] ['to', 'write', 'again', '.']
[-1, 2, -1, -1, -1, -1, -1]
```

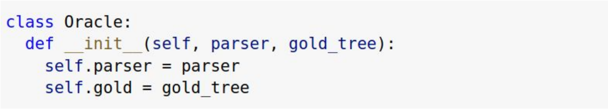
```
parser.shift()
parser.print_configuration()
```

```
['<ROOT>', 'began', 'to'] ['write', 'again', '.']
[-1, 2, -1, -1, -1, -1, -1]
```

```
parser.right_arc()
parser.print_configuration()
```

```
['<ROOT>', 'began'] ['write', 'again', '.']
[-1, 2, -1, 2, -1, -1, -1]
```

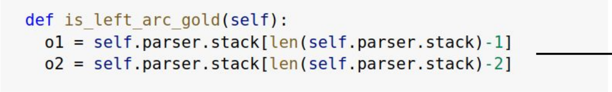
Oracle



```python
class Oracle:
  def __init__(self, parser, gold_tree):
    self.parser = parser
    self.gold = gold_tree
```

- Static
- Left-Arc precedence

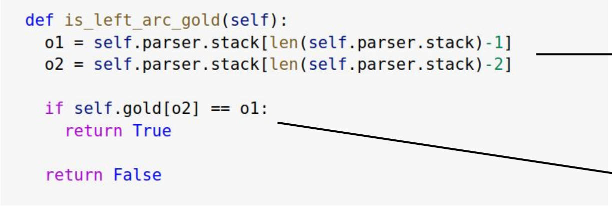
Oracle: Left-Arc



```
def is_left_arc_gold(self):
  o1 = self.parser.stack[len(self.parser.stack)-1]
  o2 = self.parser.stack[len(self.parser.stack)-2]
```

Get stack elements

Oracle: Left-Arc



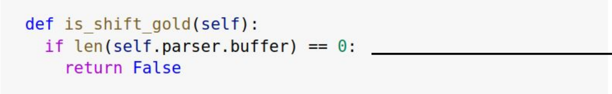
```
def is_left_arc_gold(self):
  o1 = self.parser.stack[len(self.parser.stack)-1]
  o2 = self.parser.stack[len(self.parser.stack)-2]

  if self.gold[o2] == o1:
    return True

  return False
```

Get stack elements

Verify that σ1 is parent of σ2

Note: if True, σ2 has already taken all its children because the oracle is static

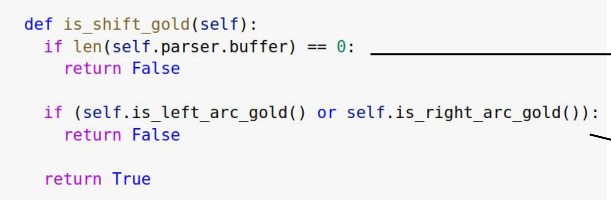
Oracle: Shift



```
def is_shift_gold(self):
  if len(self.parser.buffer) == 0:
    return False
```

Necessary condition: buffer must not be empty

Oracle: Shift



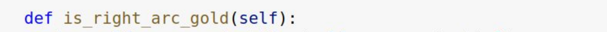
```
def is_shift_gold(self):
  if len(self.parser.buffer) == 0:
    return False

  if (self.is_left_arc_gold() or self.is_right_arc_gold()):
    return False

  return True
```

Necessary condition: buffer must not be empty

By process of elimination since the oracle is static.

Here we are implementing the Left-Arc precedence!

Oracle: Right-Arc, Your Turn!

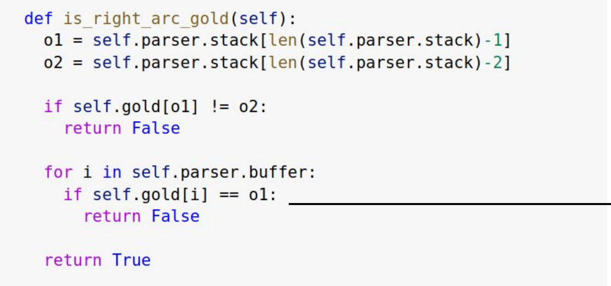


```python
def is_right_arc_gold(self):
```

?

Tip: Right-Arc must satisfy an additional condition with respect to the Left-Arc

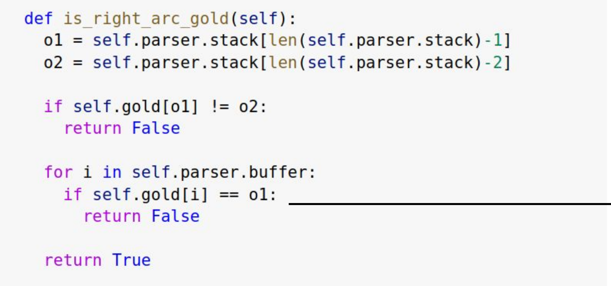
Oracle: Right-Arc



```python
def is_right_arc_gold(self):
  o1 = self.parser.stack[len(self.parser.stack)-1]
  o2 = self.parser.stack[len(self.parser.stack)-2]

  if self.gold[o1] != o2:
    return False

  for i in self.parser.buffer:
    if self.gold[i] == o1:
      return False

  return True
```

Even if σ1 is child of σ2 we must check that no children of σ1 are present in the rest of the buffer

Oracle: Right-Arc



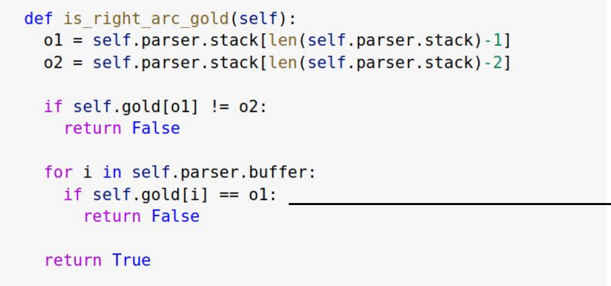
```
def is_right_arc_gold(self):
  o1 = self.parser.stack[len(self.parser.stack)-1]
  o2 = self.parser.stack[len(self.parser.stack)-2]

  if self.gold[o1] != o2:
    return False

  for i in self.parser.buffer:
    if self.gold[i] == o1:
      return False

  return True
```

Even if σ1 is child of σ2 we must check that no children of σ1 are present in the rest of the buffer

Example

b: [again, .]
s: [<ROOT>, began, write,]
a: [-1, 2, -1, 4, -1, -1, -1]
g: [-1, 2, 0, 4, 2, 4, 2]

Oracle: Right-Arc



```
def is_right_arc_gold(self):
  o1 = self.parser.stack[len(self.parser.stack)-1]
  o2 = self.parser.stack[len(self.parser.stack)-2]

  if self.gold[o1] != o2:
    return False

  for i in self.parser.buffer:
    if self.gold[i] == o1:
      return False

  return True
```

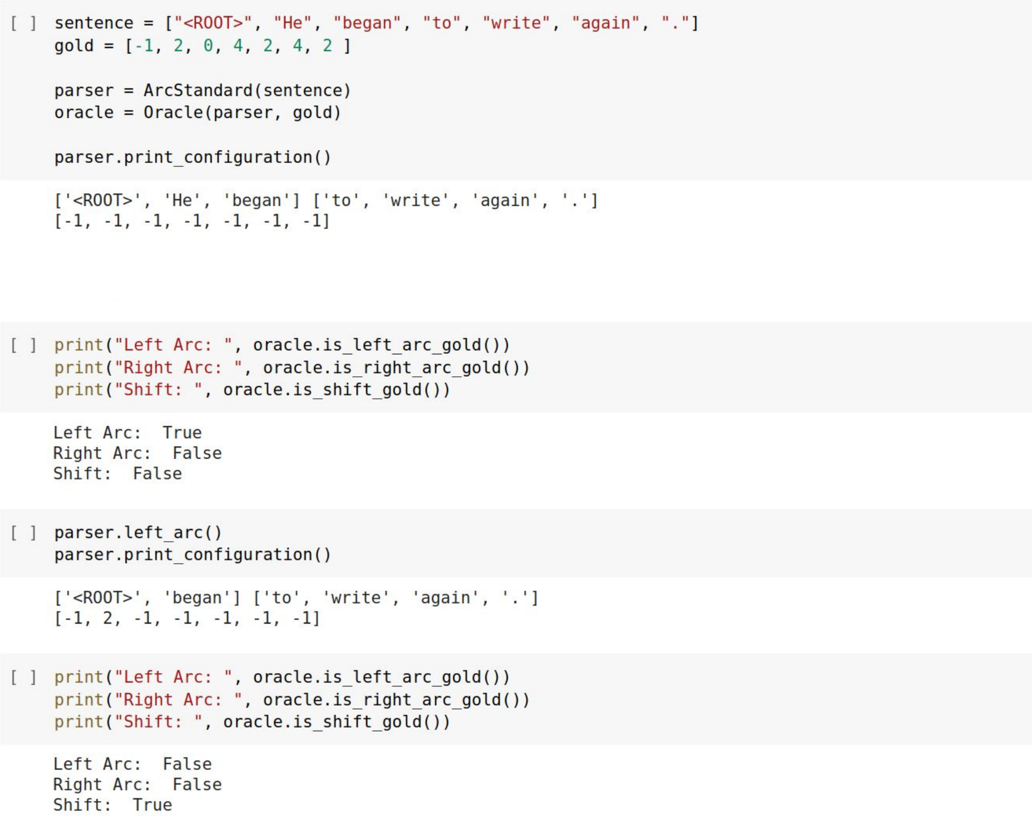
Even if σ1 is a child of σ2 we must check that no children of σ1 are present in the rest of the buffer

Example

b: [again, .]
s: [<ROOT>, began, write,]
a: [-1, 2, -1, 4, -1, -1, -1]
g: [-1, 2, 0, 4, 2, 4, 2]

write is a child of *began*, but we must wait before doing a Right-Arc otherwise we cannot attach *again* as child of *write*!

Oracle



```
sentence = ["<ROOT>", "He", "began", "to", "write", "again", "."]
gold = [-1, 2, 0, 4, 2, 4, 2 ]

parser = ArcStandard(sentence)
oracle = Oracle(parser, gold)

parser.print_configuration()
```

```
['<ROOT>', 'He', 'began'] ['to', 'write', 'again', '.']
[-1, -1, -1, -1, -1, -1, -1]
```

```
print("Left Arc: ", oracle.is_left_arc_gold())
print("Right Arc: ", oracle.is_right_arc_gold())
print("Shift: ", oracle.is_shift_gold())
```

```
Left Arc:  True
Right Arc:  False
Shift:  False
```

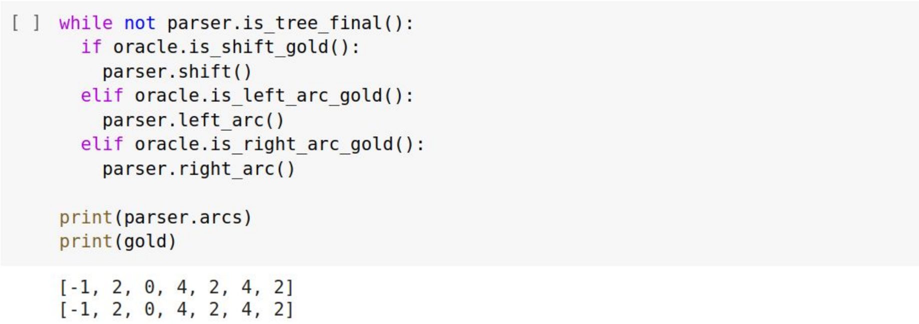
```
parser.left_arc()
parser.print_configuration()
```

```
['<ROOT>', 'began'] ['to', 'write', 'again', '.']
[-1, 2, -1, -1, -1, -1, -1]
```

```
print("Left Arc: ", oracle.is_left_arc_gold())
print("Right Arc: ", oracle.is_right_arc_gold())
print("Shift: ", oracle.is_shift_gold())
```

```
Left Arc:  False
Right Arc:  False
Shift:  True
```

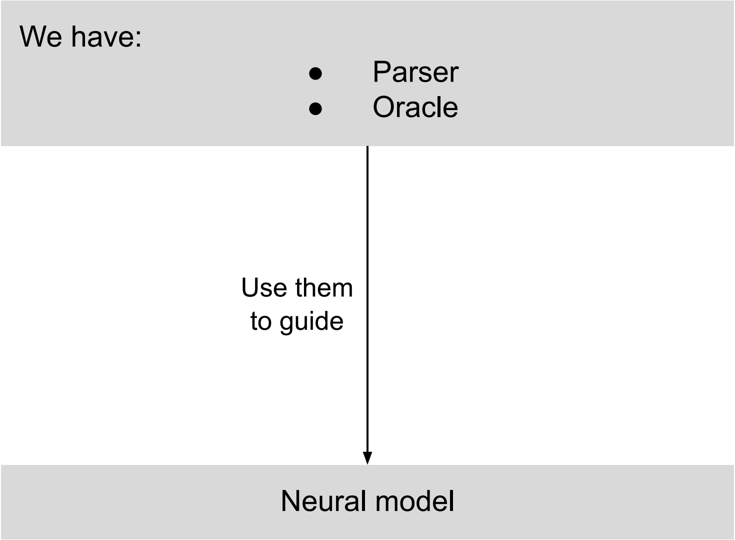
Oracle



```
while not parser.is_tree_final():
  if oracle.is_shift_gold():
    parser.shift()
  elif oracle.is_left_arc_gold():
    parser.left_arc()
  elif oracle.is_right_arc_gold():
    parser.right_arc()

print(parser.arcs)
print(gold)
```

```
[-1, 2, 0, 4, 2, 4, 2]
[-1, 2, 0, 4, 2, 4, 2]
```

Implementing a Parsing pipeline



We have:

- Parser
- Oracle

Use them to guide

Neural model

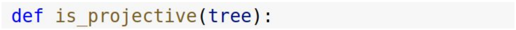
Dataset

```
dataset = load_dataset('universal_dependencies', 'en_lines', split="train")
```

```
# info about dataset
print(len(dataset))
print(dataset[1].keys())

# we look into the second sentence in the dataset and print its tokens and (gold) dependency tree
print(dataset[1]["tokens"])
print(dataset[1]["head"])
```

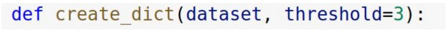
```
3176
dict_keys(['idx', 'text', 'tokens', 'lemmas', 'upos', 'xpos', 'feats', 'head', 'deprel', 'deps', 'misc'])
['About', 'ANSI', 'SQL', 'query', 'mode']
['5', '5', '2', '5', '0']
```

Dataset



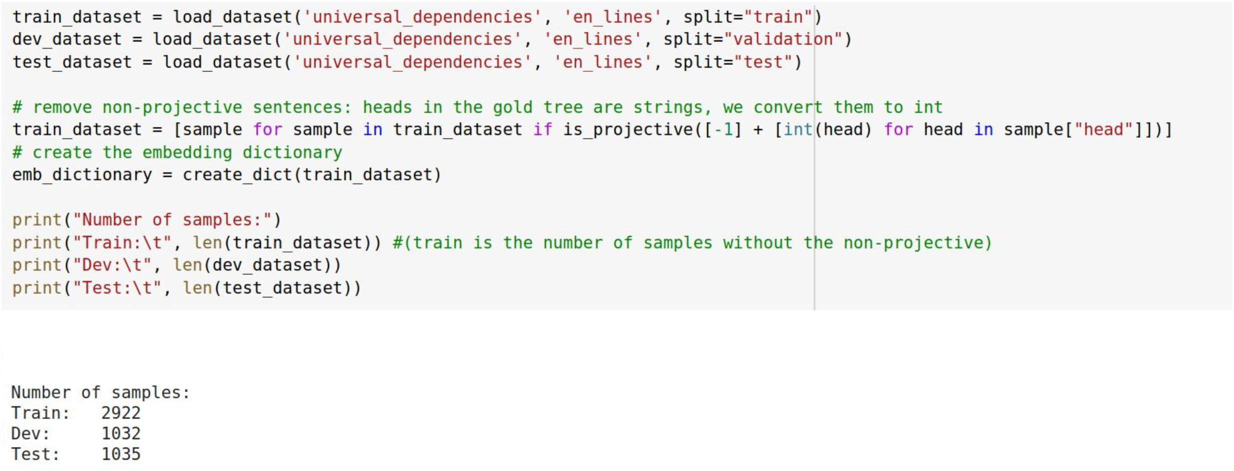
```python
def is_projective(tree):
```

Determine whether a sentence is projective



```python
def create_dict(dataset, threshold=3):
```

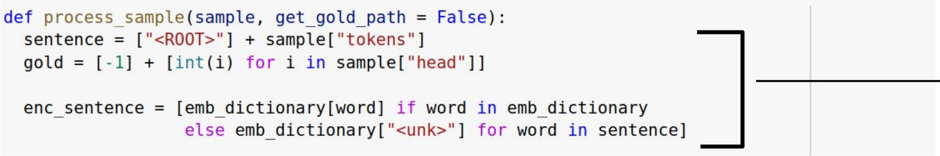
Create the word embedding dictionary



```python
train_dataset = load_dataset('universal_dependencies', 'en_lines', split="train")
dev_dataset = load_dataset('universal_dependencies', 'en_lines', split="validation")
test_dataset = load_dataset('universal_dependencies', 'en_lines', split="test")

# remove non-projective sentences: heads in the gold tree are strings, we convert them to int
train_dataset = [sample for sample in train_dataset if is_projective([-1] + [int(head) for head in sample["head"]])]
# create the embedding dictionary
emb_dictionary = create_dict(train_dataset)

print("Number of samples:")
print("Train:\t", len(train_dataset)) #(train is the number of samples without the non-projective)
print("Dev:\t", len(dev_dataset))
print("Test:\t", len(test_dataset))
```

```
Number of samples:
Train:   2922
Dev:     1032
Test:    1035
```

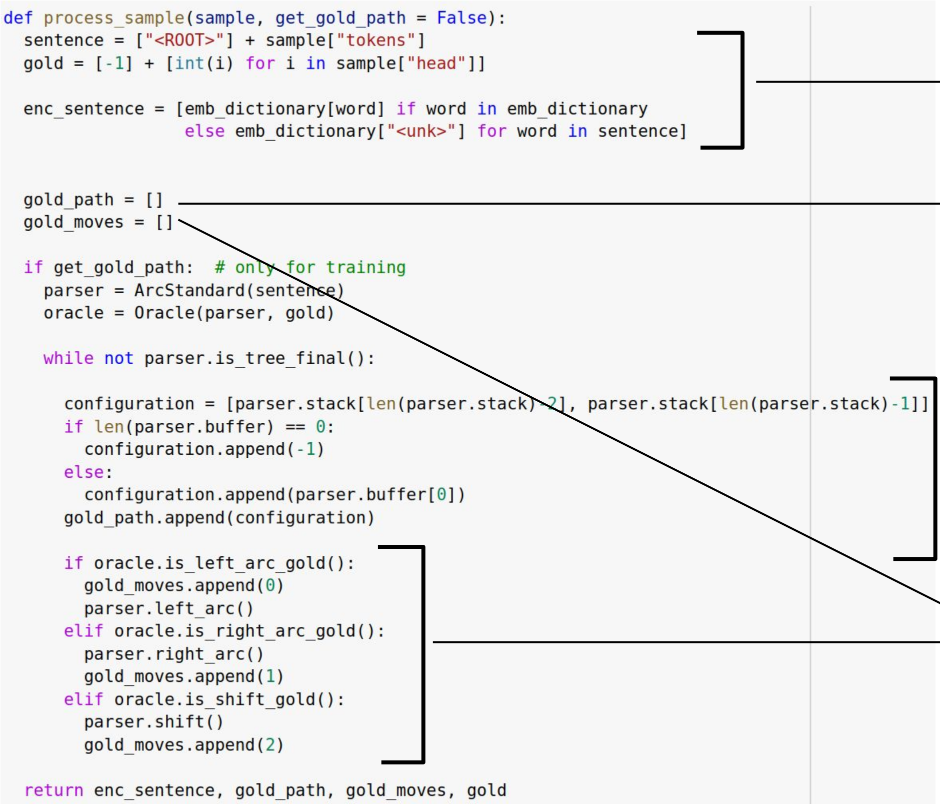
Dataloader



```
def process_sample(sample, get_gold_path = False):
  sentence = ["<ROOT>"] + sample["tokens"]
  gold = [-1] + [int(i) for i in sample["head"]]

  enc_sentence = [emb_dictionary[word] if word in emb_dictionary
                  else emb_dictionary["<unk>"] for word in sentence]
```

Initialize the sentence, the gold tree and the embeddings ids

Dataloader



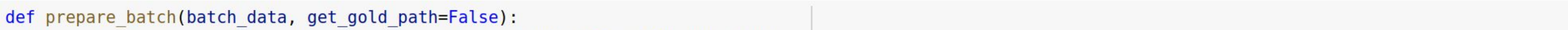
```python
def process_sample(sample, get_gold_path = False):
  sentence = ["<ROOT>"] + sample["tokens"]
  gold = [-1] + [int(i) for i in sample["head"]]

  enc_sentence = [emb_dictionary[word] if word in emb_dictionary
                  else emb_dictionary["<unk>"] for word in sentence]


  gold_path = []
  gold_moves = []

  if get_gold_path:  # only for training
    parser = ArcStandard(sentence)
    oracle = Oracle(parser, gold)

    while not parser.is_tree_final():

      configuration = [parser.stack[len(parser.stack)-2], parser.stack[len(parser.stack)-1]]
      if len(parser.buffer) == 0:
        configuration.append(-1)
      else:
        configuration.append(parser.buffer[0])
      gold_path.append(configuration)

      if oracle.is_left_arc_gold():
        gold_moves.append(0)
        parser.left_arc()
      elif oracle.is_right_arc_gold():
        parser.right_arc()
        gold_moves.append(1)
      elif oracle.is_shift_gold():
        parser.shift()
        gold_moves.append(2)

  return enc_sentence, gold_path, gold_moves, gold
```

Initialize the sentence, the gold tree and the embeddings ids

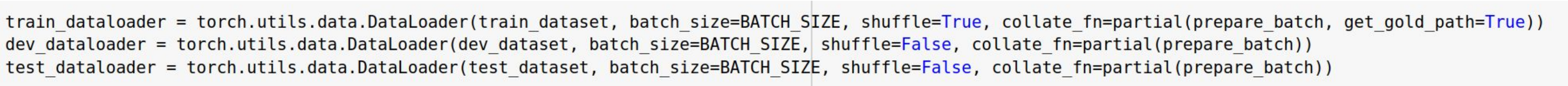
gold_path: stores the configurations of the stack and the buffer

gold_moves: stores the correct gold move at each configuration

Dataloader

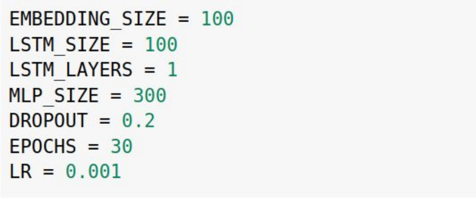


```
def prepare_batch(batch_data, get_gold_path=False):
```

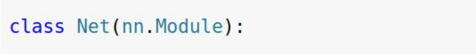


```
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=partial(prepare_batch, get_gold_path=True))
dev_dataloader = torch.utils.data.DataLoader(dev_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=partial(prepare_batch))
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=partial(prepare_batch))
```

Neural model



```
EMBEDDING_SIZE = 100
LSTM_SIZE = 100
LSTM_LAYERS = 1
MLP_SIZE = 300
DROPOUT = 0.2
EPOCHS = 30
LR = 0.001
```



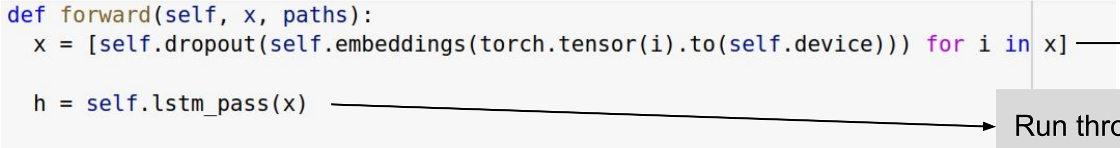
```
class Net(nn.Module):
```

Neural model

```
def forward(self, x, paths):
  x = [self.dropout(self.embeddings(torch.tensor(i).to(self.device))) for i in x]
```

Get the embeddings

Neural model

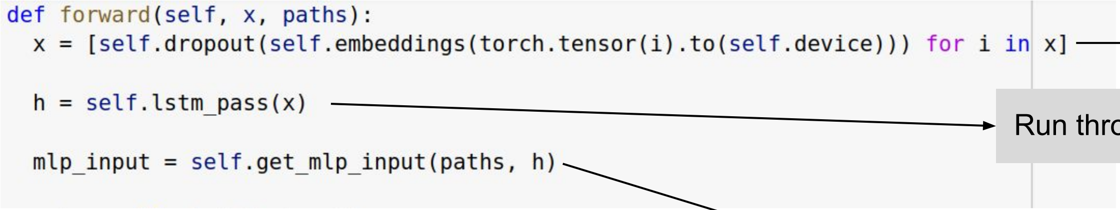


```
def forward(self, x, paths):
  x = [self.dropout(self.embeddings(torch.tensor(i).to(self.device))) for i in x]

  h = self.lstm_pass(x)
```

Get the embeddings

Run through the Bi-LSTM

Neural model



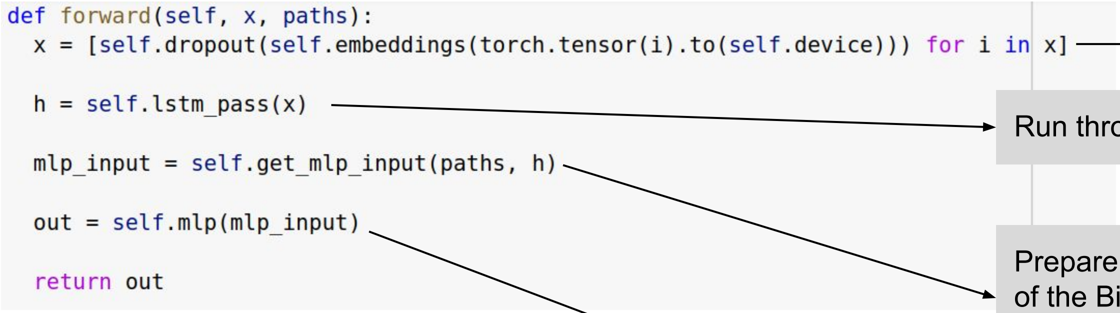
```
def forward(self, x, paths):
  x = [self.dropout(self.embeddings(torch.tensor(i).to(self.device))) for i in x]

  h = self.lstm_pass(x)

  mlp_input = self.get_mlp_input(paths, h)
```

Get the embeddings

Run through the Bi-LSTM

Prepare the input for the feedforward. Get the output of the Bi-LSTM and and prepare it according to each configuration of the parser.

Neural model



```
def forward(self, x, paths):
  x = [self.dropout(self.embeddings(torch.tensor(i).to(self.device))) for i in x]

  h = self.lstm_pass(x)

  mlp_input = self.get_mlp_input(paths, h)

  out = self.mlp(mlp_input)

  return out
```

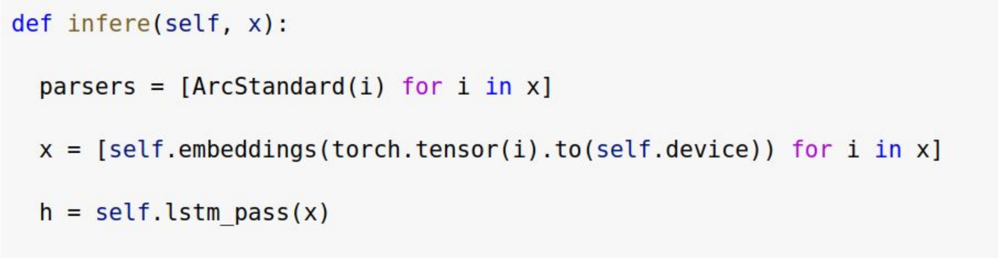
Get the embeddings

Run through the Bi-LSTM

Prepare the input for the feedforward. Get the output of the Bi-LSTM and and prepare it according to each configuration of the parser.

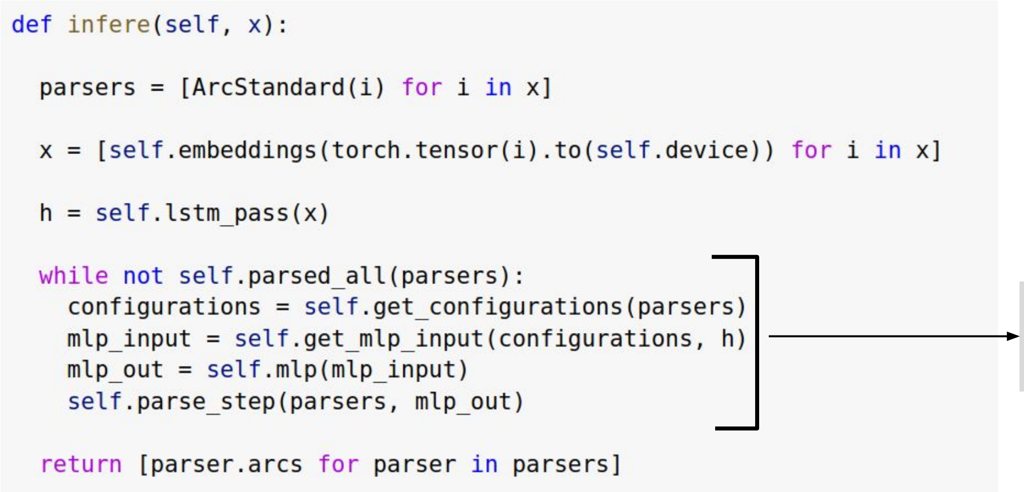
Feedforward

Neural model



```
def infere(self, x):

  parsers = [ArcStandard(i) for i in x]

  x = [self.embeddings(torch.tensor(i).to(self.device)) for i in x]

  h = self.lstm_pass(x)
```

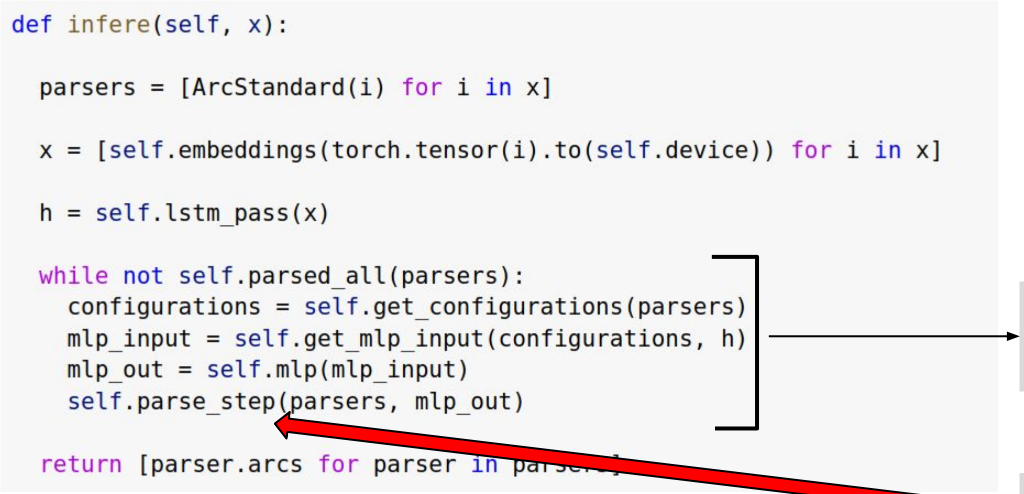
Neural model



```
def infere(self, x):

  parsers = [ArcStandard(i) for i in x]

  x = [self.embeddings(torch.tensor(i).to(self.device)) for i in x]

  h = self.lstm_pass(x)

  while not self.parsed_all(parsers):
    configurations = self.get_configurations(parsers)
    mlp_input = self.get_mlp_input(configurations, h)
    mlp_out = self.mlp(mlp_input)
    self.parse_step(parsers, mlp_out)

  return [parser.arcs for parser in parsers]
```

Inference step: the parser runs following the predictions of the model

Neural model

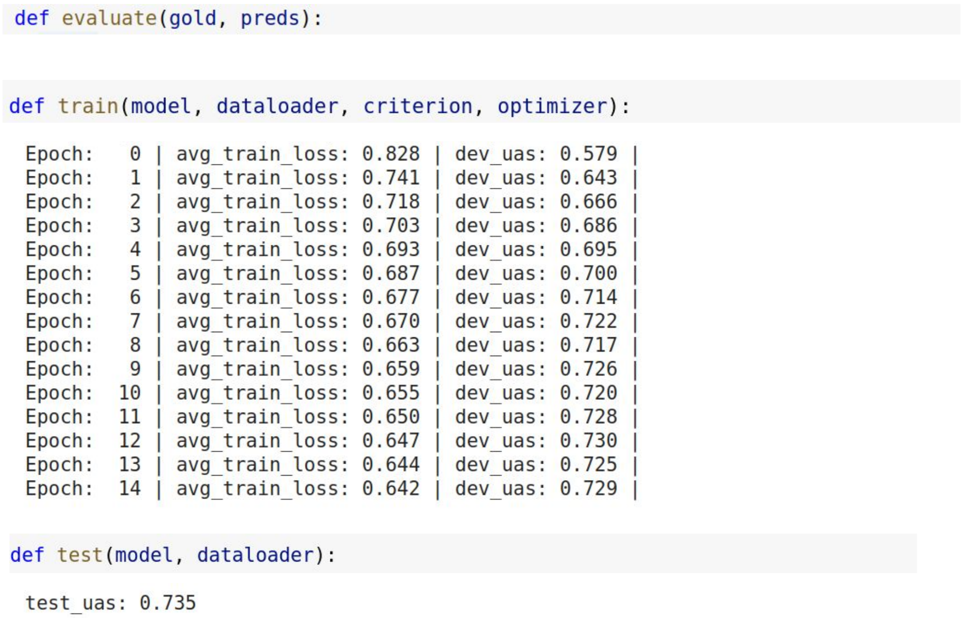


```
def infere(self, x):

  parsers = [ArcStandard(i) for i in x]

  x = [self.embeddings(torch.tensor(i).to(self.device)) for i in x]

  h = self.lstm_pass(x)

  while not self.parsed_all(parsers):
    configurations = self.get_configurations(parsers)
    mlp_input = self.get_mlp_input(configurations, h)
    mlp_out = self.mlp(mlp_input)
    self.parse_step(parsers, mlp_out)

  return [parser.arcs for parser in parsers]
```

Inference step: the parser runs following the predictions of the model

Constraints not implemented in the parser are hidden here!

Train and Test



```python
def evaluate(gold, preds):
```

```python
def train(model, dataloader, criterion, optimizer):
```

```
Epoch:   0 | avg_train_loss: 0.828 | dev_uas: 0.579 |
Epoch:   1 | avg_train_loss: 0.741 | dev_uas: 0.643 |
Epoch:   2 | avg_train_loss: 0.718 | dev_uas: 0.666 |
Epoch:   3 | avg_train_loss: 0.703 | dev_uas: 0.686 |
Epoch:   4 | avg_train_loss: 0.693 | dev_uas: 0.695 |
Epoch:   5 | avg_train_loss: 0.687 | dev_uas: 0.700 |
Epoch:   6 | avg_train_loss: 0.677 | dev_uas: 0.714 |
Epoch:   7 | avg_train_loss: 0.670 | dev_uas: 0.722 |
Epoch:   8 | avg_train_loss: 0.663 | dev_uas: 0.717 |
Epoch:   9 | avg_train_loss: 0.659 | dev_uas: 0.726 |
Epoch:  10 | avg_train_loss: 0.655 | dev_uas: 0.720 |
Epoch:  11 | avg_train_loss: 0.650 | dev_uas: 0.728 |
Epoch:  12 | avg_train_loss: 0.647 | dev_uas: 0.730 |
Epoch:  13 | avg_train_loss: 0.644 | dev_uas: 0.725 |
Epoch:  14 | avg_train_loss: 0.642 | dev_uas: 0.729 |
```

```python
def test(model, dataloader):
```

```
test_uas: 0.735
```