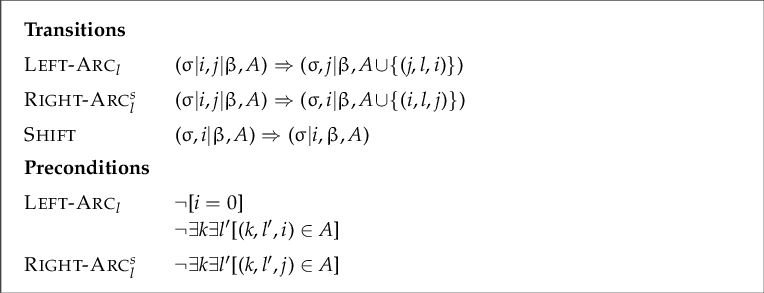
**Transition\_Parsing\_Neural Network.**

Transition-based dependency parsers (Nivre 2008) generate a dependency structure by predicting a transition action sequence. Typically, a transition system consists of a stack σ containing words being processed, a buffer β containing words to be processed and a memory A that holds the generated dependency arcs which form the partially constructed trees. A sequence of predefined transitions are incrementally produced to construct the dependency tree or graph of one sentence. At each parsing state, the next transition to be taken is either decided according to the gold structure while training or predicted by a classifier at test time



1. **Model Implementation.**

**Step 1**: Describe **class Configuration** in Configuration.py uses an indexing scheme where an index of zero refers to the ROOT node and actual word indices begin at one.

**Step 2**: data.Py script defines a transition-based dependency parser which makes use of a classifier powered by a neural network. The neural network

accepts distributed representation inputs: dense, continuous

representations of words, their part of speech tags, and the labels

which connect words in a partial dependency parse.

This is an implementation of the method described in

Danqi Chen and Christopher Manning. A Fast and Accurate Dependency Parser Using Neural Networks. In EMNLP 2014.

**Define the computational graph with necessary variables.**

self.train\_inputs = tf.placeholder(tf.int32, shape=[Config.batch\_size, Config.n\_Tokens])

self.train\_labels=tf.placeholder(tf.int32,shape=[Config.batch\_size,parsing\_system.numTransitions()])

train\_embedding\_lookup = tf.nn.embedding\_lookup(self.embeddings, self.train\_inputs)

train\_embed = tf.reshape(train\_embedding\_lookup, [Config.batch\_size, -1])

**Implement 2 Hidden Layer Neural Network.**

# **Take average loss over the entire batch**

self.loss = tf.reduce\_mean(l2\_loss)

**Test Predictions**

self.test\_inputs = tf.placeholder(tf.int32, shape=[Config.n\_Tokens])

test\_embed = tf.nn.embedding\_lookup(self.embeddings, self.test\_inputs)

test\_embed = tf.reshape(test\_embed, [1, -1])

**Step 3: Implement the forwrad pass described in**

**"A Fast and Accurate Dependency Parser using Neural Networks"(2014)**

**define forward\_pass() Function with 2 and 3 Hidden Layer.**

**Step 4 :Implement feature extraction described in**

**"A Fast and Accurate Dependency Parser using Neural Networks"(2014)**

This Method can Achieve by def getFeatures(c) Function.

**Step 5: Build the graph model**

graph = tf.Graph()

model = DependencyParserModel(graph, embedding\_array, Config)

num\_steps = Config.max\_iter

with tf.Session(graph=graph) as sess:

model.train(sess, num\_steps)

model.evaluate(sess, testSents)

**Represents a partial or complete dependency parse of a sentence, and**

**provides convenience methods for analysing the parse**

In Class **DependencyTree:**

Add the next token to the parse.

h: Head of the next token

l: Dependency relation label between this node and its head

Establish a labeled dependency relation between the two given nodes.

k: Index of the dependent node

h: Index of the head node

l: Label of the dependency relation

Get the index of the node which is the root of the parse

(i.e., the node which has the ROOT node as its head).

Check if the tree is projective using class **isProjective()**

**Step 6: Defines a transition-based parsing framework for dependency parsing**

**based on an arc-standard transition-based dependency parsing system.**

In Parsing.py Generate all possible transitions which this parsing system can

take for any given configuration.

Generate all possible transitions using **makeTransition()** which this parsing system can

take for any given configuration.

Provide a static-oracle recommendation for the next parsing step to take using def **getOracle()**

**Step 7**: Implement arc standard algorithm based on Incrementality in Deterministic Dependency Parsing(Nirve, 2004):

Left-reduce

Right-reduce

Shift

1. **Experiments.**

max\_iter = 1001

batch\_size = 10000

hidden\_size = 200

hidden2\_size = 175

hidden3\_size = 200

embedding\_size = 50

learning\_rate = 0.1

display\_step = 100

validation\_step = 200

n\_Tokens = 48

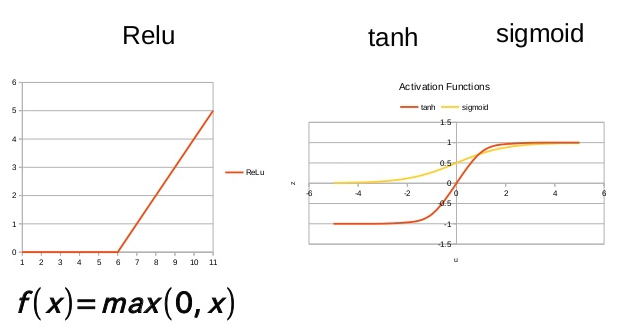
lam = 1e-8

n\_Tokens\_word = 18

n\_Tokens\_pos = 18

n\_Tokens\_labels = 12

**Several Activation Function and their Performance.**



* ***Sigmoid function →f(x) =***1/(1+ exp(-x))

It’s used in the output layer of binary classification problems which results in 0 or 1 as output. As the value of the sigmoid function lies between 0 and 1, we can predict easily to be 1 if the value is greater than 0.5 else 0. For multiclass classification problems, the Softmax function is the most common choice. The softmax function is similar to the sigmoid function

* ***Tanh function → f(x) =***2/(1 + exp(-2x))-1

It’s used in the hidden layers of neural networks as its value lies between -1 to 1, hence the mean of the hidden layer comes out to be 0 or close to 0. This helps in learning for the next layer easier.

***RELU function →f(x) =***max(0,x)

RELU is much faster than the sigmoid and Tanh function. It involves simple mathematics. If you don’t know which activation function to choose, then simply use RELUas it is the most general activation function and is used in most.

Base loss

Average loss at step 1000 : 0.363459481299

Testing on dev set at step 1000

UAS: 71.9719819528

UASnoPunc: 74.8488102639

LAS: 68.0210384625

LASnoPunc: 70.5618041033

UEM: 11.3529411765

UEMnoPunc: 11.8235294118

ROOT: 64.7647058824

Train Finished.

Starting to predict on test set

Saved the test results.

**You can see test\_predictions.conll for final output.**