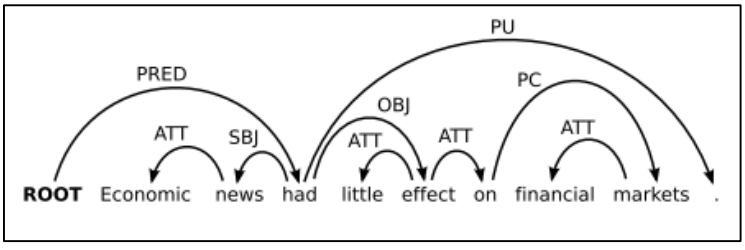
BAB II

LANDASAN TEORI

2.1 Dependency Grammar

Penggunaan *dependency grammar* diketahui sudah ada sejak abad ke-5 SM, ditandai dengan hasil karya seorang ahli bahasa bernama Pāṇini pada tata bahasa sanskerta, namun di era sekarang penggunaannya lebih mengacu pada hasil karya seorang ahli bahasa asal Perancis bernama Lucien Tesnière. *Dependency grammar* merepresentasikan *syntactic structure* melalui *dependency relation*. Setiap kata memiliki hubungan dengan kata lainnya yang sifatnya *binary asymetrical* (Kübler, dkk., 2009). *Dependency* ini memberitahukan hubungan *head-dependent* antar kata dimana sebuah kata yang menemani atau memperjelas kata lainnya disebut sebagai *dependent*, sedangkan kata yang dituju oleh *dependent* disebut sebagai *head*. Hubungan yang terjadi pada *dependent* dan *head*-nya dikelompokan berdasarkan fungsi tata bahasanya seperti subjek, predikat, objek, dsb (Kübler, dkk., 2009).



Gambar 2.1 Dependency Structure Suatu Kalimat (Kübler, dkk., 2009)

Gambar 2.1 memperlihatkan *dependency structure* untuk sebuah kalimat dalam bahasa Inggris. Dapat dilihat *dependency relation* dilambangkan oleh garis panah yang

bermula dari *head* menuju *dependent*. Selain itu setiap garis panah disertai oleh *dependency type* yang memberitahukan fungsi tata bahasanya. Contoh dari Gambar 2.1 memperlihatkan kata “news” (kata benda) adalah *dependent* dari kata “had” (kata kerja) yang mempunyai *dependency type* berupa subjek (SBJ).

2.2 Dependency Parsing

Proses mencari *dependency structure* secara otomatis oleh sistem disebut sebagai *dependency parsing*. Proses ini menerima masukan berupa kalimat yang dipecah menjadi urutan kata $w_0, w_1 \ldots w_n$. Kata w_0 adalah *ROOT* yang merupakan kata semu yang ditambahkan untuk tujuan generalisasi. Keluaran yang dihasilkan oleh *dependency parsing* ini berupa *dependency graph* yang digambarkan pada Gambar 2.1. Metode *parsing* umumnya dilakukan dengan pendekatan *data-driven* atau *grammar-based*. Metode *Grammar-based* mengandalkan *formal grammar* yang menentukan bentuk struktur suatu kalimat berdasarkan aturan tertentu, sedangkan metode *data-driven* memanfaatkan data linguistik yang sudah dianotasi untuk melatih *neural network classifier* (Kübler, dkk., 2009).

Metode *data-driven* dibagi lagi menjadi beberapa kelas salah satunya adalah *transition-based*. Metode *transition-based* merupakan suatu sistem transisi yang terdiri dari *shift-reduce transition*. Tugas *dependency parser* adalah memilih transisi yang valid secara beruntun, sehingga urutan transisi tersebut membentuk *dependency graph* dari masukan suatu kalimat (Kübler, dkk., 2009).

2.2.1 Dependency Graph

Analisis *syntactic structure* untuk suatu kalimat melalui *dependency parsing* direpresentasikan dalam bentuk *dependency graph*. Definisi formal suatu *dependency graph* ialah *labeled direted graph* yang terdiri dari himpunan simpul $V \subseteq \{w_0, w_1, ..., w_n\}$, himpunan busur $A \subseteq V \times R \times V$, dan himpunan *dependency relation* $R \subseteq \{r1, ..., rm\}$. Dengan kondisi Jika $(w_i, r, w_j) \in A$ maka $(w_i, r', w_j) \notin A$ untuk semua $r' \neq r$. R merupakan himpunan yang anggotanya terdiri atas *dependency relation types* atau *arc label* yang menerangkan fungsi tata bahasa antar dua kata dari suatu *dependency relation*. Anggota dari R diasumsikan mengikuti suatu kerangka kerja *dependency grammar* tertentu. Penelitian ini mengikuti referensi dari Universal Dependencies. Simpul V anggotanya adalah kata-kata yang dipenggal dari kalimat masukan $S = w_0 w_1 ... w_n$. Himpunan A anggotanya memberitahukan *labeled dependency relation* yang terbentuk untuk setiap kata yang ada pada V dan dengan *dependency type* r anggota dari R. Sebuah busur $(w_i, r, w_j) \in A$ diartikan sebagai *head-dependent relation* yang terjadi pada w_i selaku *head* menuju *dependent*-nya w_j beserta *dependency type* r. Aturan $(w_i, r, w_j) \in A \rightarrow (w_i, r', w_j) \notin A$ membatasi pembentukan $(w_i, r, w_j) \in A$ agar memiliki satu *dependency type* saja (Kübler, dkk., 2009). Gambar 2.2 menunjukan contoh *dependency graph* untuk Gambar 2.1

G = (V , A)

V = V_S = {root, Economic, news, had, little, effect, on, financial, markets, .}

A = {(root, PRED, had), (had, SBJ, news), (had, OBJ, effect), (had, PU, .), (news, ATT, Economic), (effect, ATT, little), (effect, ATT, on), (on, PC, markets), (markets, ATT, financial)}

Gambar 2.2 Dependency Graph dari Gambar 2.1 (Kübler, dkk., 2009)

V_S adalah *spanning node set* yang menerangkan bahwa semua kata yang ada pada masukan kalimat S = w_0 w_1 ... w_n harus menjadi bagian dari himpunan V_S.

2.2.2 Dependency Trees

Untuk setiap *dependency graph* yang memiliki *spanning node set* V = V_S dan W_0 = ROOT, maka *dependency graph* tersebut merupakan *dependency tree* (Kübler, dkk., 2009). Banyak teori dari *dependency grammar* mengharuskan *dependency graph* untuk membentuk *dependency tree* khususnya *mono-stratal dependency*. *Dependency tree* memiliki beberapa sifat lainnya, namun sebelum itu terdapat notasi yang dipergunakan untuk menjelaskan sifat tersebut yang antara lain.

1. Notasi $w_i \rightarrow w_j$ diindikasikan sebagai sebuah *dependency relation* yang ada pada *dependency tree* G = (V , A). Oleh karenanya, $w_i \rightarrow w_j$ jika dan hanya jika (w_i, r, w_j) $\in$ A untuk r $\in$ R.
2. Notasi $w_i \rightarrow^* w_j$ diindikasikan sebagai sebuah *reflexive transitive closure* dari *dependency relation* yang ada pada *dependency tree* G = (V , A). Oleh karenanya, $w_i \rightarrow^* w_j$ jika dan hanya jika i = j atau ada $w_i \rightarrow^* w_{i'}$ dan $w_{i'} \rightarrow w_j$ untuk $w_{i'} \in$ V.

Secara sederhana, bisa diartikan bahwa terdapat jalur dari w_i ke w_j melalui *dependency relation* lain yang terjadi diantara keduanya.

3. Notasi $w_i \leftrightarrow w_j$ diindikasikan sebagai sebuah *dependency relation* tak berarah pada *dependency tree* G = (V , A). Oleh karenanya, $w_i \leftrightarrow w_j$ jika dan hanya jika ada $w_i \rightarrow w_j$ ataupun $w_j \rightarrow w_i$.
4. Notasi $w_i \leftrightarrow^* w_j$ diindikasikan sebagai sebuah *reflexive transitive closure* dari *dependency relation* tak berarah yang ada pada *dependency tree* G = (V , A). Oleh karenanya, $w_i \leftrightarrow^* w_j$ jika dan hanya jika i = j atau ada $w_i \leftrightarrow^* w_{i'}$ dan $w_{i'} \leftrightarrow w_j$ untuk $w_{i'} \in V$.

Dengan hadirnya notasi diatas, sifat-sifat dari *dependency tree* dapat diperjelas sebagai berikut.

1. *Dependency tree* G = (V , A) memenuhi sifat pohon berakar, yang menyatakan tidak boleh ada $w_i \in V$ sehingga ia $w_i \rightarrow w_0$. Dengan kata lain, setiap kata tidak boleh memiliki ROOT sebagai *dependent*-nya, sehingga semua kata haruslah berasal dari ROOT.
2. *Dependency tree* G = (V , A) rentangannya mencakup semua kata pada suatu kalimat masukan, yang menyatakan $V = V_S$.
3. *Dependency tree* G = (V , A) setiap katanya saling terhubung satu sama lain, yang menyatakan, untuk semua w_i, $w_j \in V$ dan ia $w_i \leftrightarrow^* w_j$. Dapat diartikan bahwa bila arah pada *graph* dihilangkan, maka terciptalah jalur yang menghubungkan setiap kata dengan kata lainnya.

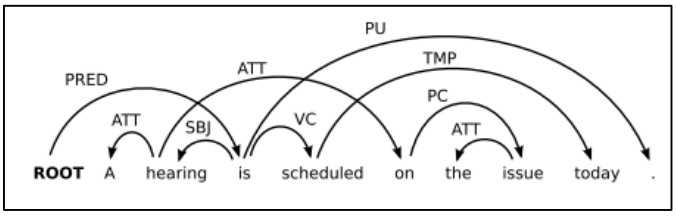
4. *Dependency tree* G = (V , A) setiap katanya hanya boleh memiliki satu *head* saja. Dinyatakan dengan, untuk semua w_i, $w_j \in V$, jika $w_i \rightarrow w_j$ maka tidak boleh ada $w_{i'} \in V$ sehingga ia $i' \neq i$ dan $w_{i'} \rightarrow w_j$.
5. *Dependency tree* G = (V , A) bersifat *acyclic*, yang berarti jalur-jalurnya tidak boleh memuat sirkuit. Dinyatakan dengan, untuk semua w_i, $w_j \in V$, jika $w_i \rightarrow w_j$ maka $w_j \rightarrow^* w_i$ tidaklah berlaku.
6. *Dependency tree* G = (V , A) mempunyai jumlah busur sebanyak |A| = |V| - 1, asalkan memiliki satu kata ROOT, dan memenuhi sifat ke-4.

Selain sifat-sifat yang telah disebutkan diatas, terdapat sifat lain yang mengharuskan *dependency tree* bersifat *projective*. Namun sifat ini tidak berlaku umum, pasalnya terdapat bahasa yang memperbolehkan penempatan kata atau frasa yang lebih bebas contohnya bahasa Turki, Belanda, dan lainnya. Terlepas dari itu, banyak *dependency parser* memiliki sifat tersebut (Kübler, dkk., 2009). Berikut ini penjelasan mengenai sifat *projective*.

1. Sebuah busur $(w_i, r, w_j) \in A$ dari suatu *dependency tree* G = (V , A) disebut *projective* jika dan hanya jika ada $w_i \rightarrow^* w_k$ untuk setiap $i < k < j$. Secara singkat dapat diartikan untuk setiap w_k yang berada diantara busur w_i dan w_j, terdapat jalur yang bisa akses oleh w_i menuju w_k.
2. Sebuah *Dependency tree* G = (V , A) disebut sebagai *projective dependency tree* apabila ia memiliki semua sifat yang telah disebutkan diatas dan semua busurnya $(w_i, r, w_j) \in A$ bersifat *projective*.

Apabila suatu *dependency tree* memenuhi sifat *projective*, maka terdapat sifat tambahan lagi yang melekat antara lain.

1. *Projective dependency tree* G = (V , A) memenuhi sifat planar suatu graf, yang menyatakan bahwa setiap busur dapat digambarkan tanpa bersinggungan dengan busur lainnya.
2. *Projective dependency tree* G = (V , A) memuat busur di dalam busur lainnya. Dinyatakan dengan, untuk semua $w_i \in V$, terdapat himpunan $\{ w_j \mid w_i \rightarrow^* w_j \}$ yang merupakan *contiguous subsequence* dari suatu kalimat masukan S. *Contiguous* memastikan bahwa untuk setiap w_j bila ditelusuri kebelakang ia akan menemui w_i.



Gambar 2.3 Dependency Tree yang Tidak Projective (Kübler, dkk., 2009)

Setelah berpanjang lebar membahas sifat-sifat *dependency tree*, maka untuk memperjelas *dependency tree* yang *projective* dan mana yang bukan masing-masing telah digambarkan pada Gambar 2.1 dan Gambar 2.3.

2.3 Transition Based Parsing

Tugas dari parsing model terbagi menjadi dua yakni *learning* dan *parsing*. *Learning* adalah proses mencari parameter yang diharapkan bisa memprediksi secara akurat *dependency tree* dari suatu kalimat. Parameter yang dimaksud didapatkan dengan metode optimisasi tertentu yang erat hubungannya dengan *machine learning*.

Setelah *learning* menghasilkan model yang dianggap mapan, model harus mampu memetakan suatu kalimat ke *dependency tree*-nya. Proses inilah yang disebut dengan *parsing* (Kübler, dkk., 2009).

Transition based parsing memodelkan kedua tugas tadi pada suatu sistem transisi. Kalimat dipetakan ke *dependency tree* dengan transisi-transisi yang terjadi pada suatu mesin abstrak, sehingga parsing bisa dilakukan. Mesin abstrak yang tadi disebut contoh konkretnya adalah *finate state automata*. Proses *learning* pada *transition based parsing* berupaya untuk memprediksi transisi apa yang harus terjadi berikutnya dengan riwayat *parsing* yang diketahui (Kübler, dkk., 2009).

2.3.1 Transition System

Sebuah sistem transisi adalah suatu mesin abstrak yang memiliki konfigurasi, dan transisi. Konfigurasi yang dimaksud mempunyai struktur internal yang berupa.

1. σ yaitu *stack* untuk menyimpan $w_i \in V_S$ yang merupakan kata-kata yang sedang diproses oleh *parser*.
2. β yaitu *buffer* untuk menyimpan $w_i \in V_S$ yang merupakan kata-kata tersisah dan menunggu untuk diproses oleh *parser*.
3. A yaitu himpunan beranggotakan $(w_i, r, w_j) \in V_S \times R \times V_S$ yang merepresentasikan suatu *dependency relation*.

Konfigurasi pada sistem transisi ini memiliki konfigurasi awal $c_0(S)$ yaitu ($[w_0]\sigma$, $[w_1,...,w_n]\beta$, ∅) untuk $S = w_0\, w_1\, ...w_n$ yang memberitahukan *stack* awalnya diisi oleh kata ROOT, *buffer* berisi semua kata pada S, dan A adalah himpunan kosong. Selain

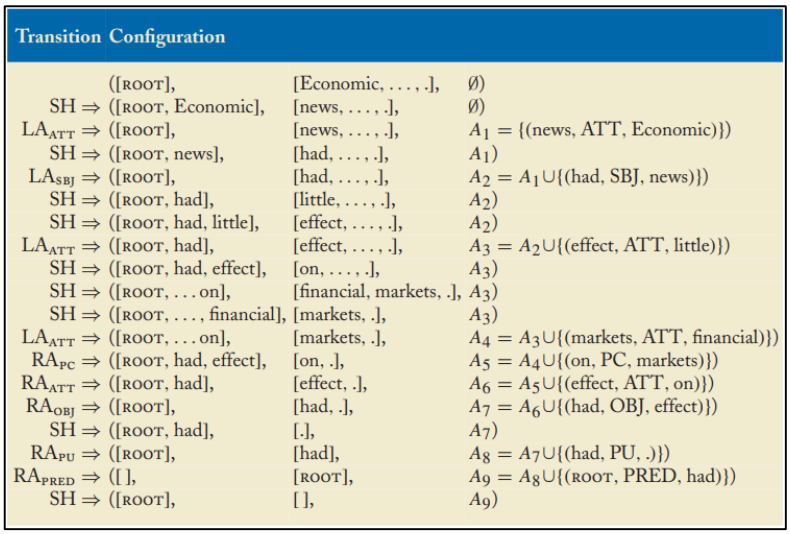
itu terdapat konfigurasi akhir yaitu (σ,[]β, A) untuk σ dan A apapun, yang diartikan transisi berakhir bila *buffer* sudah kosong (Kübler, dkk., 2009).

Setelah mendefinisikan konfigurasi diatas, sistem transisi memiliki sekumpulan transisi yang bertujuan untuk berpindah dari suatu konfigurasi ke konfigurasi selanjutnya. Transisi ini merupakan bagian dari langkah *parsing* yang diantaranya membentuk busur antarkata atau *dependency relation*, dan memodifikasi isi dari *stack* dan *buffer*. Transisi yang dimasksud diperjelas pada Tabel 2.1.

Tabel 2.1 Transisi pada *Transition Based Parsing* (Chen dan Manning 2014)

Transition	Definition	Precondition
Left-Arc	$(\sigma\|w_i\|w_j, \beta, A) \Rightarrow (\sigma\|w_j, \beta, A \cup \{(w_j, r, w_i)\})$	$\|\sigma\| \geq 2$
Right-Arc	$(\sigma\|w_i\|w_j, \beta, A) \Rightarrow (\sigma\|w_i, \beta, A \cup \{(w_i, r, w_j)\})$	$\|\sigma\| \geq 2$
Shift	$(\sigma, w_i\|\beta, A) \Rightarrow (\sigma\|w_i, \beta, A)$	$\|\beta\| \geq 1$

1. Transisi Left-Arc menambahkan *dependency relation* (w_j , r, w_i) dengan *dependency type* r kedalam himpunan busur A. w_j merupakan kata teratas pertama di *stack*, dan w_i adalah kata teratas kedua di *stack*. Kemudian buang w_i dari *stack*. Transisi ini bisa dilakukan apabila $|\sigma| \geq 2$.
2. Transisi Right-Arc menambahkan *dependency relation* (w_i , r, w_j) dengan *dependency type* r kedalam himpunan busur A. w_j merupakan kata teratas pertama di *stack*, dan w_i adalah kata teratas kedua di *stack*. Kemudian buang w_j dari *stack*. Transisi ini bisa dilakukan apabila $|\sigma| \geq 2$.
3. Transisi Shift memindahkan w_i yang adalah kata pertama di *buffer* menjadi kata teratas di *stack*, dengan kondisi $|\beta| \geq 1$.



Transition	Configuration		
	([ROOT],	[Economic, . . . , .],	Ø)
SH ⇒	([ROOT, Economic],	[news, . . . , .],	Ø)
LA_{ATT} ⇒	([ROOT],	[news, . . . , .],	$A_1 = \{(news, ATT, Economic)\}$)
SH ⇒	([ROOT, news],	[had, . . . , .],	A_1)
LA_{SBJ} ⇒	([ROOT],	[had, . . . , .],	$A_2 = A_1 \cup \{(had, SBJ, news)\}$)
SH ⇒	([ROOT, had],	[little, . . . , .],	A_2)
SH ⇒	([ROOT, had, little],	[effect, . . . , .],	A_2)
LA_{ATT} ⇒	([ROOT, had],	[effect, . . . , .],	$A_3 = A_2 \cup \{(effect, ATT, little)\}$)
SH ⇒	([ROOT, had, effect],	[on, . . . , .],	A_3)
SH ⇒	([ROOT, . . . on],	[financial, markets, .],	A_3)
SH ⇒	([ROOT, . . . , financial],	[markets, .],	A_3)
LA_{ATT} ⇒	([ROOT, . . . on],	[markets, .],	$A_4 = A_3 \cup \{(markets, ATT, financial)\}$)
RA_{PC} ⇒	([ROOT, had, effect],	[on, .],	$A_5 = A_4 \cup \{(on, PC, markets)\}$)
RA_{ATT} ⇒	([ROOT, had],	[effect, .],	$A_6 = A_5 \cup \{(effect, ATT, on)\}$)
RA_{OBJ} ⇒	([ROOT],	[had, .],	$A_7 = A_6 \cup \{(had, OBJ, effect)\}$)
SH ⇒	([ROOT, had],	[.],	A_7)
RA_{PU} ⇒	([ROOT],	[had],	$A_8 = A_7 \cup \{(had, PU, .)\}$)
RA_{PRED} ⇒	([ ],	[ROOT],	$A_9 = A_8 \cup \{(ROOT, PRED, had)\}$)
SH ⇒	([ROOT],	[ ],	A_9)

Gambar 2.4 Urutan Transisi pada *Transition Based Parsing* (Kübler, dkk., 2009)

Dependency tree suatu kalimat didapat dengan menjalankan urutan transisi untuk suatu konfigurasi awal hingga mencapai suatu konfigurasi akhir. Urutan transisi yang menyimpan *dependency tree* seutuhnya untuk suatu kalimat $S = w_0\ w_1\ \ldots\ w_n$ merupakan urutan konfigurasi $C_{0,m} = (c_0, c_1, \ldots, c_m)$ dengan c_0 sebagai konfigurasi awal, c_m sebagai konfigurasi akhir, dan untuk setiap *i* yang di rentang $1 \leq i \leq m$ terdapat suatu $t \in T$ sehingga $C_i = t(c_{i-1})$. Dengan kata lain *dependency tree* tersebut diisyaratkan berasal dari konfigurasi akhir, yang dinyatakan oleh $Gc_m = (Vs , Ac_m)$. Ac_m merupakan himpunan busur atau *dependency relation* yang ada dalam konfigurasi akhir c_m. Gambar 2.4 menunjukan urutan transisi yang terjadi untuk menghasilkan *dependency tree* utuh dari kalimat di Gambar 2.1.

2.4 Parsing Algorithm

Dependency tree dibentuk atas urutan transisi dari suatu mesin abstrak, seperti yang telah dipaparkan pada point 2.3.1. Supaya urutan transisi dapat diketahui,

diperlukan algoritma yang memilih transisi yang tidak hanya valid tetapi juga benar. Algoritma yang dimaksud ditunjukan pada Gambar 2.5.

h(S, Γ, λ)

1. $c \leftarrow c_0(S)$
2. while c is not terminal
3. $t \leftarrow \lambda_c$
4. $c \leftarrow t(c)$
5. return G_c

Gambar 2.5 Psudocode untuk Mungurai Kalimat (Kübler, dkk., 2009)

Pseudocode diatas merupakan algoritma *parsing* yang didefinisikan sebagai suatu fungsi bernama h yang menerima argumen berupa kalimat masukan S, *constraint* Γ dan parameter λ. Γ adalah sekumpulan aturan yang menetapkan *dependency tree* seperti apa yang diperbolehkan untuk suatu kalimat. Secara tidak langsung melalui pemilihan transisi yang valid, sistem transisi bertindak sebagai Γ. λ adalah parameter milik suatu *neural network classifier* yang perlu dicari dalam proses *learning*. Dengan λ diketahui, NN *classifier* dapat memprediksi suatu transisi yang kemudian dijalankan untuk berpindah ke konfigurasi berikutnya. Ketika mencapai suatu konfigurasi akhir, algoritma berhenti dan mengembalikan *dependency tree* dari kalimat S yang tersirat dalam $Gc_m = (V_S, Ac_m)$.

2.4.1 Feature Representation

Pada algoritma *parsing* yang dibahas pada point 2.4, λ_c adalah transisi yang diprediksi untuk suatu konfigurasi c menurut parameter λ dari NN *classifier*. Konfigurasi pada sistem transisi bisa punya banyak kemungkinan, sehingga diperlukan suatu langkah abstraksi agar NN *classifier* dengan mudah menerimanya sebagai masukan. Tugas mengekstrak *feature* dari suatu konfigurasi dapat didefinisikan oleh suatu fungsi f(c) : C → Y. Fungsi tersebut mengembalikan Y yang adalah *feature representation* berupa *d* dimensional vector. *Feature representation* ini biasanya bervariasi, namun penelitian ini mengacu pada *feature representation* yang digunakan pada penelitian Chen dan Manning (2014). *Indicator feature* yang mereka gunakan adalah *words*, POS *tags*, *dependency type* atau *arc* label. *Word* direpresentasikan oleh *d* dimensional $e_i^w \in \mathbb{R}^d$ yang berasal dari *embedding matrix* $E^w \in \mathbb{R}^{d \times n_w}$, dengan n_w adalah jumlah kata yang ada pada *vocabulary*. Kemudian POS *tags* dengan $e_i^p \in \mathbb{R}^d$ dan *arc label* dengan $e_i^l \in \mathbb{R}^d$ masing-masing berasal dari matriks $E^t \in \mathbb{R}^{d \times n_t}$ dan $E^l \in \mathbb{R}^{d \times n_l}$, dengan n_t dan n_l adalah jumlah POS *tags* dan label yang saling berbeda. Setelah menetapkan apa yang menjadi *indicator feature*, langkah selanjutnya adalah memilih sekumpulan *indicator* untuk suatu konfigurasi *c* yang mungkin berguna dalam proses prediksi. Pilihan tersebut diekspresikan pada himpunan S^w untuk *word*, S^t untuk POS *tags*, dan S^t untuk *arc label*. Secara lengkap anggota dari S^w berjumlah 18, yang terdiri atas.

a. Tiga kata teratas di *stack* dan *buffer* yakni s_1, s_2, s_3, b_1, b_2, b_3.

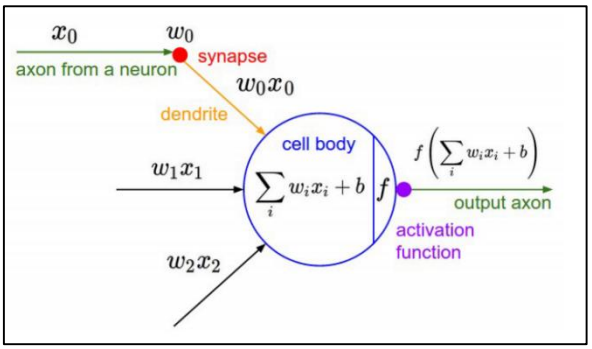
b. Anak paling kiri dan kanan yang pertama dan kedua,. untuk dua kata teratas di *stack* yakni $lc_1(s_1)$, $rc_1(s_1)$, $lc_2(s_1)$, $rc_2(s_1)$, $lc_1(s_2)$, $rc_1(s_2)$, $lc_2(s_2)$, $rc_2(s_2)$.

c. Anak paling kiri dari anak paling kiri dan anak paling kanan dari anak paling kanan, untuk dua kata teratas di *stack* yakni $lc_1(lc_1(s_1))$, $rc_1(rc_1(s_1))$, $lc_1(lc_1(s_2))$, $rc_1(rc_1(s_2))$.

Kemudian anggota dari S^t juga berjumlahkan 18 yang berisikan POS *tags* untuk setiap kata yang ada pada S^w, sedangkan S^l berjumlah 12 yang anggotanya adalah *arc label* dari setiap hubungan anak paling kiri dan kanan yang terjadi pada S^w. Ada kalanya suatu konfigurasi tidak memiliki elemen seperti apa yang telah ditetapkan pada S^w, S^t, S^l, dan ketika hal itu terjadi suatu token NULL akan digunakan untuk mengisi kekosongan tersebut.

2.5 Neural Network

Data yang melimpah dan kian cepatnya komputasi komputer saat ini memungkinkan *machine learning* khususnya *neural network* digandrungi kembali. Sebab, tidak seperti pemrograman komputer konvensional yang menuliskan setiap instruksi secara eksplisit, *neural network* sebaliknya belajar dari data untuk menemukan solusi dari suatu problem (Taylor, 2017). Unit terkecil dari *neural network* adalah *neuron* atau *node* yang ditunjukan pada Gambar 2.6.



Gambar 2.6 *Hidden Neuron* pada *Neural Network* (Manning, 2019)

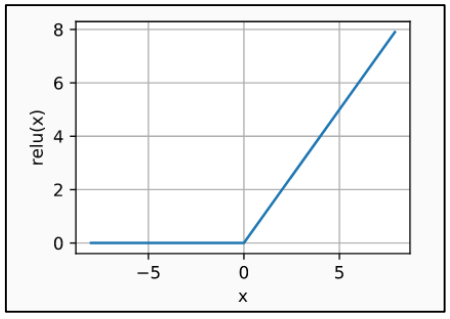
Sebuah *hidden neuron* merupakan suatu fungsi $h = f(\sum_i w_i x_i + b)$ yang menerima input x_i, *weight* w_i, bias b, berserta *nonlinear activation function* f.

2.5.1 Activation Function

Neural network membutuhkan *activation function* agar mampu menghasilkan *decision boundary* yang lebih akurat. Salah satu *activation function* yang umum digunakan adalah *rectified linear unit* (ReLU). Didefnisikan pada Persamaan 2.1 (Zhang, dkk., 2019).

$$ReLU(z) = \max(z, 0) \qquad \ldots(2.1)$$

Transformasi yang dilakukan ReLU adalah mencari nilai maksimum antara z dengan 0, sehingga nilai negatif tidak dimungkinkan semenjak z < 0 akan mengembalikan nol.

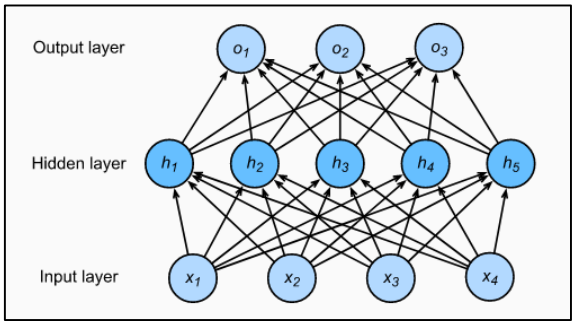


Gambar 2.7 ReLU Activation Function (Zhang, dkk., 2019)

Plot dari ReLU *activation function* ditunjukan pada Gambar 2.7. Secara jelas diperlihatkan gradiennya bernilai 1 apabila x > 0, dan bernilai 0 apabila x < 0.

2.5.2 Multilayer Perceptron

Neural network terdiri dari sekumpulan *neuron* yang saling terhubung dimaksudkan agar bisa memodelkan suatu permasalahan yang lebih kompleks. Arsitektur sederhana dari *neural network* adalah *feed-forward* yang ditunjukan pada Gambar 2.8.



Gambar 2.8 *Feed-forward Neural Network* (Zhang, dkk., 2019)

Arsitektur *feed-forward* diatas memiliki input layer dengan 4 *neuron* yang merepresentasikan *features* , *hidden layer* dengan 5 *hidden neuron*, lalu output layer dengan 3 *neuron* yang merepresentasikan kelas yang diklasifikasikan.

2.5.3 Softmax Function

Hasil keluaran dari *neural network* perlu diubah ke bentuk distribusi probabilitas agar mudah dipahami. Dengan begitu, hasil prediksi tiap kelas tidak ada yang bernilai negatif dan jika semuanya dijumlahkan akan bernilai 1. Fungsi softmax didefinisikan pada Persamaan 2.2 (Zhang, dkk., 2019).

$$\hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)} \quad \ldots(2.2)$$

2.5.4 Loss Function

Setelah *neural network* menghasilkan output yang berupa probabilitas, langkah selanjutnya adalah menghitung seberapa akurat hasil prediksi yang dikeluarkan dengan menggunakan suatu *loss function*.

$$l(y, \hat{y}) = \sum_j y_j \log \hat{y}_j \quad \ldots(2.3)$$

Cross-Entropy Loss merupakan *loss function* yang sering dipasangkan dengan softmax *classifier*. Cross-Entropy loss didefinisikan di Persamaan 2.3 (Zhang, dkk., 2019).

2.5.5 Backpropagation

Error dari *loss function* perlu diteruskan ke layer-layer sebelumnya, hal ini diperlukan untuk mengetahui gradien tiap parameter yang dibutuhkan saat melakukan parameter update. Gradien dihitung pada tiap layernya menggunakan aplikasi *chain rule*. Misalkan ada dua fungsi $Y = f(x)$, dan fungsi $Z = g(Y) = g \circ f(x)$. X, Y, Z ialah tensor. Dengan *chain rule*, turunan dari Z terhadap X dapat dituliskan seperti Persamaan 2.4 (Zhang, dkk., 2019).

$$\frac{\partial Z}{\partial X} = prod\left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}\right) \quad \ldots(2.4)$$

$prod$ adalah perkalian matriks biasa, yang mengabstraksi operasi *transpose* pada tensor.

2.5.6 Gradient Descent

Untuk bisa memperkecil error dari *loss function*, *neural network* perlu melakukan pembaharuan parameter. Pembaharuan ini dilakukan dengan gradien dari setiap parameter yang menjurus ke lokal atau global minimum dari *loss function*, sehingga diharapkan error berangsur-angsur mengecil. Algoritma inilah yang disebut *gradient descent*. Pada praktisnya, *stochastic gradient descent* yang lebih sering digunakan. Dia merupakan salah satu varian dari *gradient descent* yang melakukan pembaharuan parameter untuk setiap *mini-batches example*. *Training* data dibagi lagi menjadi beberapa tumpukan yang lebih kecil yang disebut dengan *mini-batch*.

$$(w, b) \leftarrow (w, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(w,b)} l^{(i)}(w, b) \qquad \ldots(2.5)$$

Stochastic gradient descent ditunjukan pada Persamaan 2.5 (Zhang, dkk., 2019). Gradien setiap parameter (weight dan bias) dirata-rata terhadap jumlah batch size $\mathcal{B}$, lalu dikalikan dengan suatu *learning rate* η, setelah itu parameter diperbaharui dengan mengurangi parameter lama dengan gradien yang dihitung pada *mini-batch* tadi. Variable learning rate η dan batch-size $\mathcal{B}$, termasuk *hyper-parameters* yang harus dicari manual melalui proses trial dan error, proses mencarinya disebut *hyper-parameters tuning*. Proses mencari *learning rate* melibatkan proses yang siknifikan. Oleh karenanya terdapat metode lain dari *gradient descent* yang mampu secara adaptif men-*tuning learning rate* ini dengan sendirinya. Salah satunya adalah Adam yang dijadikan *default optimizer* dalam algoritma *gradient descent learning*.

$$m \leftarrow \beta_1 m + (1 - \beta_1) \nabla_\theta J_{mini-batch}(\theta) \quad \ldots(2.6)$$

$$v \leftarrow \beta_2 v + (1 - \beta_2) \left(\nabla_\theta J_{mini-batch}(\theta) \odot \nabla_\theta J_{mini-batch}(\theta)\right) \quad \ldots(2.7)$$

$$\theta \leftarrow \theta - \alpha \odot m\sqrt{v} \quad \ldots(2.8)$$

Pada Persamaan 2.6-2.8 (Kingma dan Ba, 2014), θ adalah vektor yang menyimpan semua model parameter (wight dan bias), α adalah *learning rate, J* adalah *loss function*, dan $\nabla_\theta J_{mini-batch}(\theta)$ adalah gradien dari *loss function* terhadap semua model parameter pada suatu *mini-batch*. Adam mempergunakan teknik momentum yang dihitung oleh *m*, momentum ini menyebabkan nilai update memiliki *low variance,* sehingga proses *learning* bisa lebih cepat. Selain itu Adam menghitung *v* yaitu *adaptive learning rates* untuk setiap parameter, hal ini bertujuan agar parameter yang jarang diperbaharui akan menerima nilai perbaharuan yang lebih besar dibanding dengan parameter yang lebih sering diperbaharui. β_1 dan β_2 adalah *hyper-parameter*, tapi biasanya ditetapkan mengikuti *default value*-nya yaitu 0.9 dan 0.99 masing-masing.

2.6 ELMo

ELMo adalah model *neural network* yang terlatih untuk melakukan tugas pemodelan bahasa. ELMo mempunyai fungsi dan internal *state* yang menerima kalimat masukan untuk menghasilkan representasi kata. Karena fungsi dalam ELMo menerima kalimat bukan hanya kata untuk menghasilkan representasi kata maka representasi ini bersifat konteks dependen. Fungsi dalam ELMo menerima kata yang direpresentasikan sebagai kumpulan vektor karakter sehingga menjadikannya sebagai model berbasiskan karakter atau *character-based model*.

Struktur ELMo terdiri atas tiga lapisan. Lapisan paling bawah ialah CNN yang berfungsi merepresentasikan kata berdasarkan karakter, dua lapisan di atasnya ialah BiLSTM yang berfungsi menghasilkan representasi kata yang bersifat konteks sensitif. Setiap representasi yang dihasilkan setiap lapisannya menyimpan informasi kata yang berbeda. Lapisan teratas yakni BiLSTM dinilai lebih menyimpan informasi semantik. Pada lapisan terbawah yakni CNN dinilai lebih menyimpan informasi sintaksis dan morfologis.

ELMo mengekspos representasi kata dari ketiga lapisan. Penggunaan representasi kata bisa secara individu pada tiap lapisan atau dengan kombinasi linear ketiga lapisan yang ditunjukan pada persamaan 2.9.

$$ELMo_k^{task} = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \, \mathrm{h}_{k,j}^{LM} \qquad \ldots(2.9)$$

2.6.1 RNN Language Models

Tujuan dari *language model* adalah memperkirakan distribusi probabilitas dari urutan kata atau suatu kalimat. *Language model* yang baik akan menghasilkan probabilitas tinggi untuk suatu kalimat yang sering didapati dalam penggunaan bahasa. *Language model* yang baik juga dapat melakukan prediksi kata secara tepat. *Language models* yang dikembangkan ELMo merupakan *Recurrent Neural Network* (RNN) based *language model* yang menggunakan varian dari RNN yaitu *Long Short-Term Memory* (LSTM). Untuk menghitung *hidden state* di suatu *timestep t* LSTM menetapkan mekanisme *gating* yang bertujuan untuk mengingat *long-term dependency*

atau riwayat input. Mekanisme *gating* ini dilakukan dengan menambahkan sel memori dan beberapa *gate* yang ditunjukan pada persamaan 2.10-2.15.

$$i_t = \sigma\left(W^i x_t + U^i h_{t-1} + b^i\right) \quad \ldots(2.10)$$

$$f_t = \sigma\left(W^f x_t + U^f h_{t-1} + b^f\right) \quad \ldots(2.11)$$

$$o_t = \sigma\left(W^o x_t + U^o h_{t-1} + b^o\right) \quad \ldots(2.12)$$

$$\tilde{c}_t = \tanh(W^c x_t + U^c h_{t-1} + b^c) \quad \ldots(2.13)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad \ldots(2.14)$$

$$h_t = o_t \odot tanh(c_t) \quad \ldots(2.15)$$

Langugae model yang dikembangkan ELMo merupakan *bidirectional language model* (biLM). Alasan ELMo dibilang biLM adalah ia mampu memprediksi kata dengan mengetahui konteks yang muncul sebelum (*forward*) atau sesudah (*backward*) kata yang akan diprediksi. Sebagai contoh, diberikan suatu kalimat yang terdiri dari kata $(t_1, t_2, \ldots, t_N)$ sejumlah N. Probabilitas kalimat tersebut dihitung dengan konteks sebelah kiri ditulis pada persamaan 2.16.

$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, t_2, \ldots, t_{k-1}) \quad \ldots(2.16)$$

Sedangkan probabilitas kalimat dengan menggunakan konteks dari sebelah kanan ditulis pada persamaan 2.17

$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1}, t_{k+2}, \ldots, t_N) \quad \ldots(2.17)$$

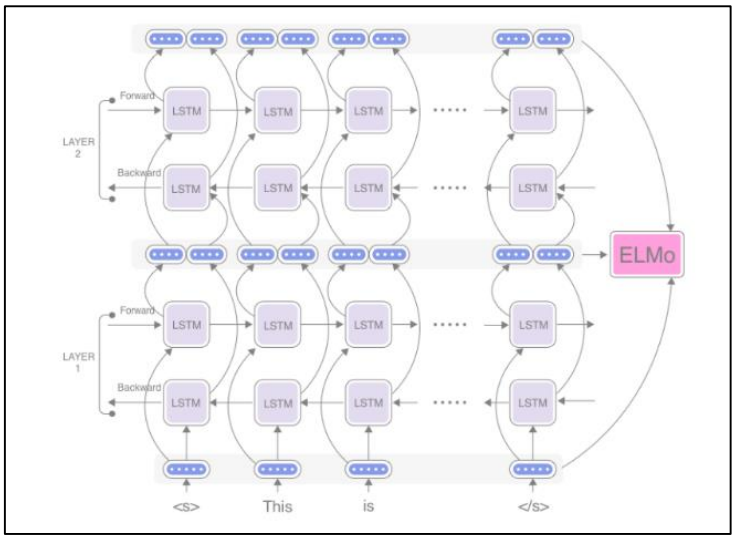
Probabilitas dari masing-masing kata dapat dihitung pada persamaan 2.18

$$p(t_k = j \mid t_1, t_2, \dots, t_{k-1}) = \frac{\exp\left(h_t\, p^j + q^j\right)}{\sum_{j' \in V} \exp\left(h_t\, p^{j'} + q^{j'}\right)} \qquad \dots(2.18)$$

p^j dan q^j masing-masing adalah weight dan bias dari *neuron* ke- j pada output layer. ELMo menggunakan *stacked* LSTM yang artinya layer LSTM bisa lebih dari satu. ELMo memiliki dua layer LSTM sehingga untuk setiap *timestep* t memiliki *hidden state* $\vec{h}_{t,j}^{LM}$ untuk *forward* dan *backward* $\overleftarrow{h}_{t,j}^{LM}$ dengan $j = 1, 2$. *Hidden state* yang digunakan pada persamaan adalah *hidden state* teratas dengan $j = 2$. Fungsi objektif dari ELMo adalah memaksimalkan probabilitas kata yang benar dari *forward* dan *backward language model* yang ditulis pada persamaan 2.19.

$$\sum_{k=1}^{N} \Big(\log p\left(t_k \mid t_1, \dots, t_{k-1};\ \Theta_x, \vec{\Theta}_{LSTM}, \Theta_s\right) + \log p\left(t_k \mid t_{k+1}, \dots, t_N;\ \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s\right) \Big) \qquad \dots(2.19)$$

Parameter-parameter yang dipelajari dari biLM adalah Θ_x untuk representasi kata, Θ_s untuk softmax layer, dan $\vec{\Theta}_{LSTM}$, $\overleftarrow{\Theta}_{LSTM}$ untuk *forward backward* LSTM.



Gambar 2.9 Struktur Internal ELMo (Hagiwara, 2018)

Gambar 2.9 memperlihatkan bagaimana biLM di dalam ELMo dibangun. Proses dimulai pada layer CNN yang menghasilkan vektor kata, kemudian vektor kata tersebut diteruskan ke layer LSTM yang pertama, diteruskan lagi ke layer LSTM yang kedua. Terakhir *hidden state* dari layer LSTM diteruskan ke layer output (softmax) untuk menghasilkan prediksi kata.

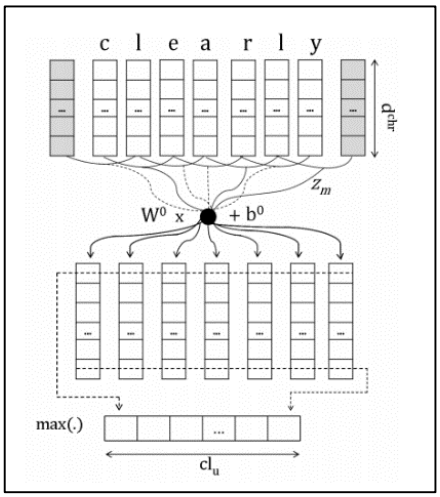
2.6.2 Character CNN Representation

ELMo menggunakan CNN untuk menghasilkan representasi kata tingkat karakter. Untuk mengetahui representasi kata tingkat karakter r^{wch} suatu kata w hal pertama yang dilakukan adalah memecah kata w menjadi kumpulan karakter $\{c_1 c_2, \ldots, c_M\}$. Setiap karakter lalu diubah ke bentuk vektor dimana untuk suatu karakter c_m vektornya adalah r_m^{chr}. Suatu vektor r_m^{chr} dari karakter m didapat dengan melakukan operasi *dot product* terhadap *one-hot encoding* v^m dengan matriks *embedding* W^{chr}. CNN menerima masukan berupa deretan karakter vektor

$\{r_1^{chr} r_2^{chr}, \ldots, r_M^{chr}\}$. Suatu jendela konvolusi sebesar k^{chr} ditetapkan untuk menentukan area konvolusi $z_m = \left(r^{chr}_{m-(k^{chr}-1)/2}, \ldots, r^{chr}_{m+(k^{chr}-1)/2}\right)^T$. Selanjutnya representasi kata tingkat karakter r^{wch} dihitung dengan persamaan 2.20.

$$[r^{wch}]_j = \max_{1<m<M} \left[w^0 z_m + b^0\right]_j \quad \ldots(2.20)$$

Proses konvolusi melakukan *dot product* terhadap weight matrix w^0 (*kernel*) dengan area konvolusi z_m. Kemudian diikuti dengan *max pooling layer* (dos Santos dan Zadrozny, 2014).



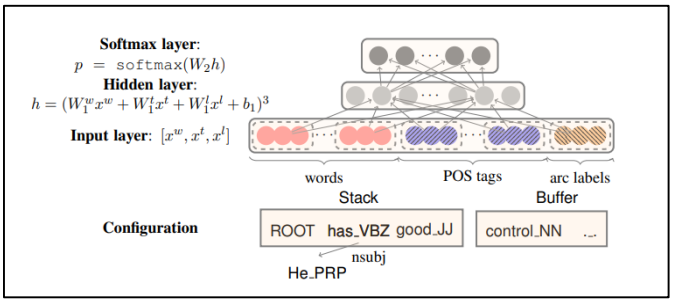
Gambar 2.10 Konvolusi Karakter (dos Santos dan Zadrozny, 2014)

Gambar 2.10 menampilkan keseluruhan proses dalam menghasilkan representasi kata tingkat karakter menggunakan CNN.

2.7 FNN Based Dependency Parser

Untuk dapat melakukan penguraian dependensi secara baik diperlukan suatu fungsi *oracle*. Fungsi *oracle* adalah fungsi yang menerima masukan berupa konfigurasi

dan mengembalikan suatu transisi dengan tepat. Tugas dari FNN adalah sebagai *classifier* yang mengaproksimasikan fungsi *oracle* tersebut. Struktur internal dari FNN terdiri dari input layer, satu *hidden layer*, dan output layer. *Forward pass* diawali dengan mengubah *feature* yang didapat dari *feature extraction* (subbab 2.4.1) ke bentuk *embeddings*. *Feature* tersebut adalah *word embeddings*, POS *embeddings*, label *embeddings*. *Word embeddings* berasal dari representasi kata tingkat karakter milik ELMo, POS dan label *embedding* masing-masing diinisiasi secara acak untuk dijadikan parameter yang dipelajari model. Ketiga *embeddings* tersebut disambungkan sehingga *input feature* yang diterima oleh classifier ialah $x = [e^w; e^p; e^l]$. *Input feature* tersebut lalu diteruskan ke hidden layer untuk menghitung *hidden state* $h = ReLu(W_1 x + b)$. h diteruskan ke output layer untuk menghitung probabilitas $p = softmax(W_2 h)$. Dari distribusi probabilitas p diambil transisi $t = \arg\max p$ dengan nilai probabilitas tertinggi. Proses training dimulai dengan menyiapkan *traning set example* sebanyak m yang terdiri atas *tuple* berisikan konfigurasi c dan transisi t yakni $\{(c_i, t_i)\}_{i=1}^{m}$. Fungsi objektif dari FNN adalah $L(\theta) = -\sum_i \log p_{t_i}$ yang meminimalkan nilai *negative log likelihood* dari transisi t yang benar.



Gambar 2.11 Struktur Internal FNN (Chen dan Manning, 2014)

Struktur internal dari FNN ditunjukan pada Gambar 2.11. Model pada Gambar 2.11 mempergunakan *cube activation function* sedangkan penelitian ini mempergunakan ReLU *activation function*. Selebihnya struktur FNN di penelitian ini sama dengan Gambar 2.11.

2.7.1 Learning Algorithm

FNN dapat dilatih dengan *gradient decent learning* yang menggunakan algoritma *backpropagation*. Proses *backpropagation* menghitung gradien terhadap error untuk tiap parameter dari output hingga ke input layer. *Backpropagation* dimulai dengan menghitung gradien error pada output layer yaitu $\frac{\partial L}{\partial o} = \hat{y} - p$.

$\hat{y}$ adalah target transisi dan p adalah prediksi transisi dari FNN. Dengan melakukan *chain rule* maka gradien dari W_2 menjadi $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial w_2} = \frac{\partial L}{\partial o}\, h$. Selanjutnya gradien dari h adalah $\frac{\partial L}{\partial h} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h} = \frac{\partial L}{\partial o}\, w_2$. Setelah menghitung semua gradien di output layer, selanjutnya hitung gradien di *hidden layer*. Mula-mula gradien dari *activation* ReLu $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial z} = \frac{\partial L}{\partial h}\, ReLu'(z)$. Lanjut ke gradien W_1 yakni $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = \frac{\partial L}{\partial z}\, x$. Kemudian gradien b yakni $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial b} = \frac{\partial L}{\partial z}$. Terkahir gradien x yakni $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial L}{\partial z}\, W_1$. Setelah mengetahui semua gradien dari parameter, langkah selanjutnya adalah melakukan parameter update menggunakan Adam *optimizer*.

2.8 Universal Dependencies

Universal Dependencies (UD) adalah pengembangan lanjutan dari representasi Stanford Dependencies (SD). Kedua representasi ini fungsinya adalah

mendeskripsikan hubungan gramatik dalam kalimat (sintak) melalui *typed dependency relation*. Contoh relasi dependensi dari kalimat "Budi pergi ke Sekolah" yaitu kata "Budi" memiliki hubungan gramatik sebagai subjek dari kata "pergi". Di dalam relasi dependensi, kata "Budi" disebut *head* dan kata "pergi" disebut *dependent*. SD dikembangkan secara khusus untuk bahasa Inggris, oleh karena itu pengembangan UD ditujukan untuk penyederhanaan, penyeragaman representasi SD agar representasi UD bisa diaplikasikan pada bahasa lain. UD memiiki sekitar 40 hubungan gramatik yang berlaku umum untuk bermacam-macam bahasa. Namun tidak semua hubungan gramatik tersebut dipergunakan dalam mendeskripsikan sintak bahasa Indonesia. Hubungan gramatik yang dipergunakan dalam bahasa Indonesia dapat dilihat pada Tabel 2.2.

Tabel 2.2 *Dependency Relation* dari *Indonesian Treebank* (McDonald, dkk., 2018)

Label	**Definisi**	**Contoh**
acl : clausal modifier of noun (adjectival clause)	Klausa ajektiva yang menerangkan kata nomina	"Bau apakah yang tercium olehku ?" → acl(Bau, tercium)
advcl : adverbial clause modifier	Klausa adverbia yang menerangkan kata predikat	"Beberapa hari kemudian ia menyatakan menyerah secara damai" → advcl(menyatakan, hari)
advmod : adverbial modifier	kata atau frasa adverbia yang menerangkan makna suatu kata	"Pinisi sebenarnya merupakan nama layar" → advmod(merupakan, sebenarnya)
amod: adjectival modifier	Kata atau frasa ajektiva yang menerangkan kata nomina atau pronomina	"Liburan yang indah mendadak menjadi penuh darah" → amod (Liburan, indah)
appos: appositional modifier	Kata nomina yang mendefinisikan kata nomina lain.	"Apakah engkau memperhatikan hamba-Ku

		Ayub" → appos(hamba-Ku, Ayub)
aux: auxiliary	Kata kerja bantu yang terasosiasi dengan kata kerja utama	"Mariah harus beristirahat dari tampil di depan umum" → aux(beristirahat,harus)
case: case marking	Element untuk menandakan preposisi, klitik.	"Dibawah Pimpinan beliau SMA Negeri 3 berkembang" → case(Pimpinan,Dibawah)
cc: coordinating conjunction	Relasi antara kata hubung dengan kata yang diperhubungkan	"Stres atau penyalahgunaan zat ?" → cc(penyalahgunaan, atau)
ccomp: clausal complement	Klausa dependent yang berfungsi sebagai objek dari kata kerja atau sifat	"Dia mengatakan kamu suka berenang" → ccomp(mengatakan,suka)
compound: compound	Penggabungan dua kata yang membentuk kata majemuk	"Pengeluaran baru ini dipasok oleh rekening bank gemuk Clinton" → compound(rekening, bank)
compound:plur	Sub-tipe dari compound yang mendeskripsikan reduplikasi kata	"Sebagian besar dari teks - teks ini ditulis setelah abad ke – 11" → compound:plur(teks,teks)
conj: conjunct	Relasi antara kata-kata yang diperhubungkan dengan kata hubung	"Jenis / merk apa ?" → conj (Jenis, merk)
cop: copula	Kata kerja bantu yang menghubungkan subjek dengan predikat non verba	"Let It Shine adalah film televisi 2012" → cop(film,adalah)
csubj: clausal subject	Klausa yang menjadi subjek dari klausa lain	"Pada 16 Juli berita adanya pembantaian mulai tersebar" → csubj(tersebar, adanya)
csubj:pass	Sub-tipe dari csubj yang adalah subjek dari klausa bentuk pasif	"berjenis - jenis ulat diketahui sebagai hama yang rakus" → csubj:pass(diketahui, berjenis)
dep: unspecified dependency	Relasi yang tidak teridentifikasi. Bisa karena penggunaan tata bahasa yang salah	
det: determiner	Kata yang memperjelas kata atau frasa nominal dengan memberikannya referensi jarak, kepunyaan, kuantitas.	"Masjid ini terbesar di India" → det(Masjid,ini)

fixed: fixed multiword expression	Satu kesatuan kata yang terdiri dari dua kata atau lebih yang sering digunakan bersama untuk menyampaikan suatu ekspresi gramatik	"John Hick adalah salah satu tokoh yang menggunakan pandangan ini" → fixed(salah,satu)
flat: flat multiword expression	Satu kesatuan kata yang terdiri dari dua kata atau lebih untuk mengekspresikan penamaan, penulisan tanggal.	"Let It Shine adalah film televisi 2012" → flat(Let,it), flat(it,shine)
iobj: indirect object	Kata nomina yang menjadi objek tidak langsung dari kata verba	"Maukah kalian aku ceritakan kisah tentang Abu Dzar ?" → iobj(ceritakan,kalian)
mark: marker	Kata penanda yang menghubungkan klausa dependen dengan klausa independen	"Namun , masih ada pertanyaan yang tak terjawab" → mark(terjawab, yang)
nmod: nominal modifier	kata nomina yang menerangkan kata atau frasa nominal lain.	"Film ini terdiri dari 6 musim dengan 137 episiode" → nmod(musim, episiode)
nsubj: nominal subject	kata atau frasa nominal yang menjadi subjek dari klausa	"Liburan yang indah mendadak menjadi penuh darah" → nsubj(mendadak,Liburan)
nsubj:pass	Sub-tipe dari nsubj yang adalah subjek dari klausa bentuk pasif	"Sebelumnya , jet tersebut hanya dilihat oleh blogger" → nsubj:pass(dilihat, jet)
nummod: numeric modifier	Kata bilangan yang menerangkan kuantitas kepada kata nomina	"Periode tunggu saat ini adalah delapan minggu" → nummod(minggu, delapan)
obj: object	Kata atau frasa nominal yang menerima aksi dari kata verba	"Konsumen bisa melejitkan permintaan untuk perubahan" → obj(melejitkan, permintaan)
obl: oblique nominal	Kata nomina, pronomina, frasa nominal yang berfungsi sebagai kalimat pelengkap bagi kata verba, adjektiva, adverbia.	"Apakah yang engkau serukan kepada kami ?" → obl(serukan, kami)
parataxis: parataxis	Relasi yang menghubungkan frasa atau klausa yang saling independen namun tidak disertai dengan kata hubung atau konjungsi	"Lagi pula , internet bukan barang mewah ; internet adalah alat penting" →parataxis(barang,alat)

punct: punctuation	Mengindikasikan tanda baca	“Toh saya tetap akan masuk penjara , semoga sepadan" → punct(sepadan, ‘,’)
root: root	Mengindikasikan akar kalimat. *Head* dari relasi ini adalah token semu yang bernama ROOT.	“Toh saya tetap akan masuk penjara , semoga sepadan" → root(ROOT, masuk)
xcomp: open clausal complement	predikat atau klausa komplemen dari kata verba atau kata adjektiva. xcomp tidak memiliki subjek internal.	“Philip kemudian bergerak melawan musuh - musuh nya di selatan” → xcomp(bergerak, melawan)