BAB 2
TINJAUAN PUSTAKA

2.1 Kajian Terkait

Beberapa kajian terkait berikut secara langsung menjadi dasar dalam penelitian ini, terutama Sennrich dkk (2016) dengan "*Edinburgh Neural Machine Translation Systems for WMT16*". Penelitian tersebut mendasari arsitektur terhadap sistem *Nematus*, yang mana contoh training dengan *MarianNMT* dibuat berdasarkan dari sistem utama tersebut. Di penjelasan dalam *repository* MarianNMT terutama contoh konfigurasi-nya menamakan model ini sebagai *Nematus-Style Shallow RNN*, yang mana sistem *Nematus* pada aslinya memiliki spesifikasi berbeda dan tentunya dengan skala lebih besar. Model utama yang digunakan berupa *seq2seq* dengan *bidirectional* RNN. Arah penerjemahan dari bahasa Romania ke Inggris dengan beberapa sub-tugas, seperti penerjemahan dengan panjang kalimat bervariasi menggunakan *toolkit Nematus*. Tersebutlah di dalam file contoh, yang mana penggunaan modul tambahan *sentencepiece* (Kudo & Richardson, 2018) berpengaruh besar untuk *pipeline* terbarunya yang memberi nilai BLEU sebesar 36.5(dev) dan 35.1(devtest) melewati skor sistem aslinya sebanyak 2 poin dengan toolkit penerusnya, MarianNMT. Skor BLEU otomatis didapat dari modul skoring otomatis *SacreBLEU* (Post, 2018).

Penelitian yang dijalankan (Gunawan et al., 2021) memiliki beberapa notasi penting yang mana dapat memberi pengertian lebih mendalam terkait jenis dan pengaplikasian mekanisme *attention*, serta dasar representasi data tiap tahapnya dalam NMT. Pengenalan *pipeline* NMT dengan jumlah kata korpus yang kecil serta mekanisme *teacher forcing* dalam decoder juga ditekankan disini. Evaluasi model dan penerjemahan menggunakan *10-fold cross validation* untuk meminimalkan efek masalah OOV(*out-of-vocabulary*) pada pelatihannya. Perangkat pemrosesan berbasis *cloud service* digunakan dan dijelaskan disini untuk menghilangkan batasan dari device lokal.

Penelitian (Wdowiak, 2021) untuk pengembangan penerjemah mesin dalam Bahasa Sicilian ke Bahasa Inggris memberikan poin penting untuk pengembangan NMT dengan kondisi *low-resource*, terutama untuk kondisi paralel dataset yang susah ditemukan. Sumber data diperoleh dengan menggunakan

terjemahan dari buku dan artikel tertentu secara manual dan juga data melalui monolingual text yang diproses dengan backtranslation untuk mendapatkan pasangannya. Penelitian dilakukan dengan self-attentional neural network (Vaswani et al., 2017), dan penggunaan fitur seperti subword-splitting, dropout, dan self-attention memberikan efek besar untuk kasus terjemahan dua arah untuk kasus Bahasa diatas. Skor BLEU berkisar antara 20-30, yang mana pada kasus multilingual dapat mencapai skor diatas 30.

Penelitian (Sennrich & Zhang, 2019) mendeteksi pada awal ditemukannya masalah terkait NMT yang memiliki kualitas yang jatuh drastis pada kondisi *low-resource*, dan berkinerja lebih buruk daripada *phrase-based statistical machine translation* (PBSMT) dan membutuhkan jumlah data tambahan yang besar untuk mendapatkan hasil yang kompetitif. Dengan *Nematus*, dan dengan berbagai optimasi yang dilakukan untuk korpus TED dari IWSLT 2014 untuk bahasa Jerman ke Inggris. Untuk kasus data dengan kondisi *low-resource*, dilakukan *data cleanup* untuk training dan dev split dari (Ranzato et al., 2015), mendapat 159K kalimat paralel untuk *training*, dan 7584 untuk *development*. Training akan dilakukan dengan data tersebut dan optimasi dilakukan seperti 'reduce BPE vocabulary', 'reduce batch size' dapat mendorong peningkatan skor BLEU sampai 75% dari skor yang didapat dengan optimasi biasa dalam model *Nematus*. Skor berurut dari model biasa, 'reduce BPE', dan 'reduce batch size': (7.20 ± 0.62, 12.10 ± 0.16, 12.40 ± 0.08).

Adapun perbandingan penelitian yang dilakukan dapat dilihat pada Tabel 2.1.1.

Tabel 2.1.1 Tabel Perbandingan Penelitian

No.	Penulis	Judul	Keterangan
1.	(Sennrich et al., 2016b)	*Edinburgh Neural Machine Translation Systems for WMT16*	- Encoder-Decoder dengan Attention (Bahdanau et al., 2015). - ensemble *decoding* dan pervasive dropout - Ukuran minibatches 80, panjang kalimat maximum 50, ukuran word embeddings 500, ukuran hidden layers 1024. - gradient norm ke 1.0 - training model dengan Adadelta

			- validasi model setiap 10000 minibatches update dengan BLEU pada validation set (newstest2013, newstest2014, atau setengah dari newsdev2016 untuk EN↔RO) - early stopping untuk model tunggal - penggunaan 4 last saved models (dengan models disimpan setiap 30000 minibatches) untuk ensemble results. - *Decoding* dilakukan dengan beam search dengan beam size 12. - menggunakan AmuNMT C++ decoder - membangun neural translation systems untuk 4 pasangan bahasa: English↔Czech, English↔German, English↔Romanian dan English↔Russian - menggunakan BPE subword segmentation untuk terjemahan open-vocabulary dengan fixed vocabulary - back-translations otomatis dari korpus monolingual News sebagai tambahan training data
2.	(Gunawan et al., 2021)	Analisis Perbandingan Nilai Akurasi Mekanisme *Attention* Bahdanau dan Luong pada *Neural Machine Translation* (NMT) Bahasa Indonesia ke Bahasa Melayu Ketapang dengan Arsitektur *Recurrent Neural Network* (RNN)	- Encoder-Decoder dengan RNN GRU. - Attention Bahdanau dan Luong. - Cross-entropy loss. - Pelatihan dengan teacher forcing. - Pengujian dengan k-fold cross validation, jumlah epoch, dan manual (ahli bahasa). - Sumber data 5000 kalimat. - Penilaian dilakukan dengan BLEU untuk mengukur akurasi terjemahan dalam persen.

3.	(Wdowiak, 2021)	*Sicilian Translator: A Recipe for Low-Resource NMT*	- Self-attentional network (Transformer) (Vaswani et al., 2017) - Sockeye toolkit untuk training - BPE atau subword splitting - Layer ke 3, ukuran embedding ke 256, ukuran model ke 256, attention head ke 4, dan feedforward ke 1024 - Parameter dropout yang dimaksimalkan - 260K kata dengan 16K baris kalimat dalam korpus sumber dan target. - Skor BLEU diperoleh untuk penerjemahan searah dengan pertambahan token kata atau baris kalimat (10 - 30).
4.	(Sennrich & Zhang, 2019)	*Revisiting Low-Resource Neural Machine Translation: A Case Study*	- NMT dengan Nematus, PBSMT dengan Moses - Data korpus TED dari IWSLT 2014 - *Low-resource* dengan 100K kata - *Data cleanup* dan *train/dev split* (Ranzato et al., 2015) - Optimasi *hyperparameter* dasar Nematus - Representasi *subword*, BPE - Hyperparameter tuning, untuk *batch size* - Lexical Model (Nguyen & Chiang, 2018) - Skor tertinggi pada kasus *low-resource* didapat dengan optimasi normal Nematus, ‘reduce BPE vocabulary’, ‘reduce batch size’, ‘aggressive word dropout’, dan ‘other hyperparameter tuning’. Proses bertahap menghasilkan skor BLEU (16.57±0.26).

Adapun penjelasan penelitian yang akan dilakukan dapat dilihat pada Tabel 2.1.2.

Tabel 2.1.2 Penelitian yang Dilakukan

No.	Penulis	Judul	Keterangan
1	Dzulkahfi	Analisis Hasil Training dan Penerjemahan Terhadap Mesin Penerjemah Syaraf MarianNMT dari Bahasa Inggris ke Indonesia	-Korpus paralel dengan 230K-450K kalimat dari bahasa Inggris ke bahasa Indonesia. -Evaluasi dengan "FLORES-101"offline. -Implementasi NMT dengan Toolkit MarianNMT -Pipeline dasar dari "*Nematus-Style Shallow RNN*" dengan *Sentencepiece* -.Skoring menggunakan BLEU dan spBLEU yang didapat secara otomatis dengan modul SacreBLEU

Berdasarkan studi literatur yang telah disampaikan dapat disimpulkan bahwa penelitian ini akan mengimplementasikan mesin penerjemah jaringan saraf tiruan (NMT) dengan model "Nematus-Style Shallow RNN" di *toolkit* MarianNMT. Penelitian ini menghasilkan tujuan untuk mengetahui kinerja dan kualitas penerjemahan dari *toolkit* MarianNMT melewati hasil skoring di beberapa prosesnya. Penelitian akan dikhususkan dengan model diatas, dan akan mencoba beberapa kasus terhadap kondisi data dalam korpus training maupun validasi. Arah bahasa dari Inggris>Indonesia, dan dataset untuk validasi akan mengambil dari FLORES-101 offline.

2.2 Pemrosesan Bahasa Alami

Natural Language Processing atau disingkat NLP menurut (Liddy, 2001) adalah suatu fokus Artificial Intelligence yang bertujuan untuk menciptakan pemrosesan bahasa layaknya manusia. Dalam mewujudkannya dibutuhkan analisis terhadap kumpulan data yang berbentuk kata. Semua dilakukan dengan berbagai teknik komputasi yang memungkinkan penyelesaian suatu tugas atau aplikasi tertentu.

NLP memiliki banyak bidang pemahaman yang bermacam-macam, contohnya seperti Linguistik yang fokus ke struktur model dari bahasa dan penemuan lingkungannya; Sains yang mengarah ke representasi internal dari data dan pemrosesan struktur yang efisien; Psikologi yang melihat bahasa sebagai proses

kognitif manusia, dan memodelkan bahasa sesuai untuk penggunaan di bidangnya. Tiap disiplin ilmu mengalamatkan permasalahannya secara unik, sehingga memberi ragam permasalahan yang berbeda-beda.

Divisi fokus dalam NLP tidak mendasar sesuai namanya, melainkan dibagi menjadi dua sebagai language processing (pemrosesan bahasa) dan language generation (penghasil bahasa). Proses pertama merupakan analisis dari bahasa dengan tujuan menghasilkan representasi yang bermakna, sedangkan terakhir untuk memproduksi bahasa tersebut untuk disampaikan dari representasi proses sebelumnya.

NLP dapat mencakup ke berbagai macam tugas, metode dan fenomena linguistik. Dalam pernyataan Eisenstein (2018), ada beberapa perspektif khusus untuk mendeskripsikan elemen penting dalam bahasa tersebut, baik itu lewat kata, kalimat atau bahkan suara. *Relational* memfokuskan proses analisis terhadap kaitan asal muasal atau rantai hubungan tiap elemen dalam bahasa untuk mencapai pemahamannya. *Compositional* mengefektifkan panjang dan susunan elemen, yang mana nantinya akan dipecah menjadi bagian-bagian dasar dan dianalisis tiap bentuk pecahan tersebut untuk membentuk pemahaman dari tiap bagiannya. *Distributional* mengefektifkan pengenalan pola dalam beberapa contoh kemunculan elemen bahasa dengan bentuk yang sama, yang ditentukan dari derajat kemunculan dengan kasus yang sama di bentuk berbeda.

Menjembatani pemahaman antara manusia dan mesin bukanlah hal yang mudah, yang mana tiap bagian penelitian yang diimplementasikan tidak harus sepenuhnya diaplikasikan, terutama dengan berbagai perspektif berbeda diatas. Ketiga pendekatan perspektif tersebut dapat berkontribusi penting dalam pengenalan secara linguistik, bahkan menjadi kritis untuk kasus dalam NLP. Dengan penyatuan ketiga perspektif itu kemungkinan menciptakan solusi terbaik untuk NLP dapat dipastikan.

2.3 Mesin Penerjemah Jaringan Saraf Tiruan

Neural Machine Translation atau disingkat NMT datang dari berbagai usaha implementasi ulang dari *neural network* ke dalam penerjemahan mesin. Yang mana (Stahlberg, 2019), menyimpulkan beberapa prosesnya sebagai pendekatan

untuk menggunakan neural network yang diaplikasikan sebagai komponen dalam sistem Statistical Machine Translation. Dengan menetapnya model kombinasi log-linear dan hanya mengubah beberapa bagian dalam arsitektur terdahulunya.

NMT mendapat posisinya dengan implementasi *Single Large Neural Network* yang secara langsung men-transformasi kalimat asal ke kalimat tujuan. NMT menggunakan *continuous representation*, berbeda dengan sistem SMT yang menggunakan *discrete symbolic representation*. Semua itu didapat dengan konsep pelatihan *end-to-end*, yang bahkan dapat mencapai kondisi performa *state-of-art* di berbagai pasangan bahasa.

Performa NMT yang didorong dengan pendekatan terhadap kondisi data dalam penerjemahan mesin, juga memperhatikan aplikasi terhadap pendekatan probabilistik sebagai satu bagian. (Tan et al., 2020) meyakini itu sebagai tujuan dari sistem NMT, yang digambarkan dalam rumus matematis $P = (y|x)$ pada dataset D dimana, *x* dan *y* adalah variabel acak yang mewakili *input* sumber dan *output* target.

(Tan et al., 2020) mengklasifikasikan garis besar dari konsep NMT menjadi tiga bagian, yaitu *Modeling*, *Inference*, dan *Learning*.

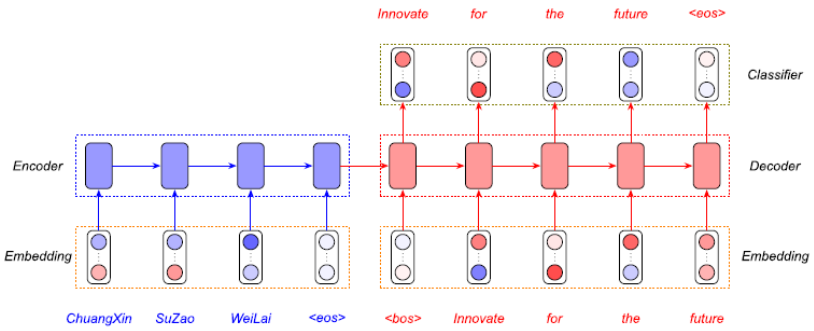
2.3.1 Modeling

Penerjemahan dapat dimodelkan pada tingkatan yang berbeda, seperti dokumen, paragraf atau kalimat. Pada penjelasan ini fokus pada tingkatan kalimat dalam penerjemahan. Selain itu, menggunakan kalimat *input* sumber dan *output* target sebagai *sequence* atau urutan. Demikian model NMT dapat dilihat sebagai model *sequence to sequence*. Dengan asumsi kita berikan kalimat sumber $x = \{x1, x2, \dots, xn\}$ dan kalimat target $y = \{y1, y2, \dots, yn\}$. Dengan menggunakan chain rule, aturan *conditional distribution* dapat difaktorkan dari kiri kekanan (L2R), seperti pada Persamaan 2.1.

$$P(\boldsymbol{y} = \mathbf{y}|\boldsymbol{x} = \mathbf{x}) = \prod_{t=1}^{T} P(y_t|y_0, \dots, y_{t-1}, x_1, \dots, x_S). \quad (2.1)$$

Hampir semua model mesin penerjemah jaringan saraf tiruan menggunakan *framework encoder-decoder* (Cho et al., 2014). *encoder-decoder framework* ini terdiri dari empat komponen dasar yaitu *embedding layer*, *encoder*

network, decoder network dan *classification layer*, Seperti pada Gambar 2.3.1.



Gambar 2.3.1 Overview dari arsitektur NMT

Embedding layer mewujudkan prinsip *continuous representation*. Hal ini memetakan simbol diskrit x_t menjadi *continuous vector* $x_t \in \mathbb{R}^d$, dimana *d* menunjukan dimensi vektor. Kemudian dimasukan kedalam lapisan selanjutnya untuk *feature extraction*.

Encoder network memetakan *embedding source* ke *hidden continuous representation*, untuk mempelajari *expressive representation*. *Encoder* harus dapat memodelkan bahasa sumber dengan *recurrent neural network* (RNN), dengan komputasi seperti pada Persamaan 2.2.

$$\mathbf{h}_t = \mathrm{RNN}_{\mathrm{ENC}}(\mathbf{x}_t, \mathbf{h}_{t-1}). \tag{2.2}$$

Dengan menerapkan *state transition function* $\mathrm{RNN}_{\mathrm{ENC}}$ pada urutan *input*, kita dapat menggunakan *final state* h_s sebagai representasi untuk seluruh kalimat sumber, dan kemudian memasukkannya ke *decoder*.

Decoder network dapat dilihat sebagai *language model* berkondisi pada pada h_s. *Decoder network* mengekstrak informasi yang diperlukan dari *encoder output* dan memodelkan dependensi jarak antara kata target.

Diberikan simbol awal y_0 = <sos> dan *initial state* $s_0 = h_s$, RNN pada *decoder* mengkompres *decoding history* pada $\{y_0, \ldots, y_{t-1}\}$ menjadi *state vector* $s_t \in \mathbb{R}^d$, seperti pada Persamaan 2.3.

$$\mathbf{s}_t = \mathrm{RNN}_{\mathrm{DEC}}(\mathbf{y}_{t-1}, \mathbf{s}_{t-1}). \tag{2.3}$$

Classification layer memprediksi distribusi dari token target. Classification layer biasanya merupakan layer linier dengan *softmax activation function*. Dengan asumsi kosa kata bahasa target adalah v dan $|v|$ adalah ukuran dari kosa kata. Diberikan *decoder output* $s_t \in \mathbb{R}^d$, *classification layer* pertama-

tama memetakan **h** ke vektor **z** didalam ruang vocabulary $\mathbb{R}^{|v|}$ dengan peta linier. Kemudian *softmax function* digunakan untuk memastikan *output vector* adalah probabilitas yang valid seperti pada Persamaan 2.4.

$$\text{softmax}(\mathbf{z}) = \frac{\exp(\mathbf{z})}{\sum_{i=1}^{|V|} \exp(\mathbf{z}_{[i]})}, \tag{2.4}$$

Merujuk pada Persamaan 2.4, dimana kita menggunakan $z_{[i]}$ untuk menyatakan komponen ke-i dalam **z**.

2.3.2 Inference

Diberikan model NMT dan x sebagai kalimat sumber, bagaimana menghasilkan a sebagai terjemahan dari model adalah masalah penting. Idealnya kita akan menemukan y sebagai kalimat target dengan memaksimalkan model prediksi $P = (y|x = x; \theta)$ sebagai terjemahan. Namun, karena ukuran ruang pencariannya yang sangat besar, tidak praktis untuk menemukan terjemahan dengan probabilitas tertinggi. Oleh karena itu, NMT biasanya menggunakan algoritma pencarian seperti *greedy search* atau *beam search* untuk menemukan terjemahan terbaik.

2.3.3 Training Model NMT

NMT biasanya menggunakan *maximum log-likelihood* (MLE) sebagai *training object function* yang merupakan metode yang umum digunakan untuk memperkirakan parameter *probability distribution*. Secara formal, memberikan *training set* $D = \{\langle x^{(s)}, y^{(s)} \rangle\}_{s=1}^{S}$, tujuan dari *training* untuk menemukan satu set parameter model yang memaksimalkan *log-likelihood* pada *training set* seperti pada Persamaan 2.5.

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \underset{\boldsymbol{\theta}}{\text{argmax}} \left\{ \mathcal{L}(\boldsymbol{\theta}) \right\}, \tag{2.5}$$

Dimana *log-likelihood* didefinisikan sebagai Persamaan 2.6.

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{s=1}^{S} \log P(\mathbf{y}^{(s)}|\mathbf{x}^{(s)}; \boldsymbol{\theta}). \tag{2.6}$$

Berdasarkan algoritma *back-propagation* kita dapat secara efisien menghitung gradien $\mathcal{L}$ terhadap θ. Pelatihan model NMT biasanya mengadopsi

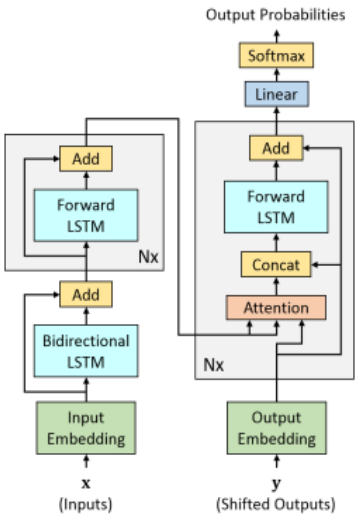
algoritma *stochastic gradient search* (SGD). Dari pada menghitung *gradient* pada semua *training set*, SGD menghitung *lose function* dan *gradient* pada *minibatch training set*. SGD optimizer memperbarui parameter model NMT dengan aturan seperti pada Persamaan 2.7.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}), \tag{2.7}$$

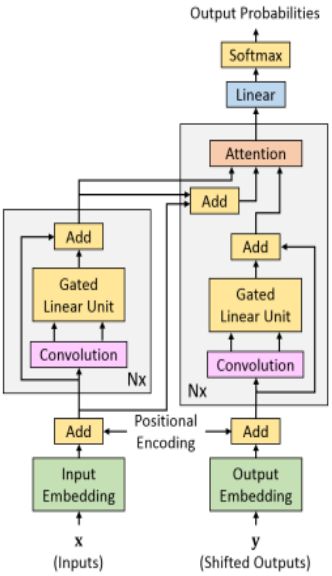
Dimana α adalah *learning rate*. Dengan *learning rate* yang dipilih dengan baik, parameter NMT menjamin dicapainya *local optima*. Dalam praktiknya, selain penggunaan SGD biasa, *adaptive learning rate* seperti Adam (Kingma & Ba, 2015) dapat mengurangi waktu *training* s.

2.3.4 Arsitektur Mesin Penerjemah Jaringan Saraf Tiruan

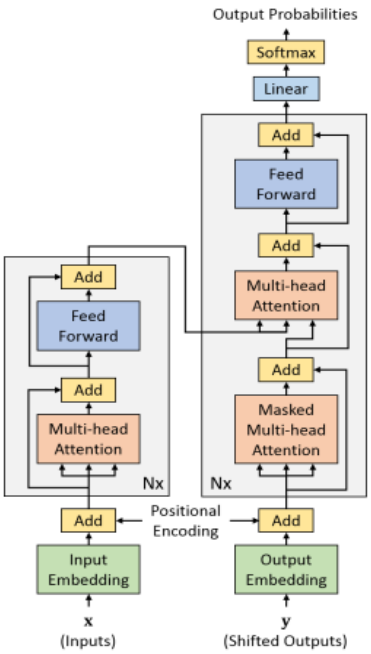
NMT memiliki tiga jenis fundamental arsitektur yang umum digunakan, antara lain berdasar *reccurent*, *convolutional*, dan *self-attention*. Gambar 2.3.2 menunjukkan aliran data dalam sistem *Google's Neural Machine Translation* (Wu et al., 2016) sebagai contoh *reccurent network*, model *convolutional ConvS2S* (Gehring et al., 2017), dan basis model *self-attention Transformer* (Vaswani et al., 2017) secara berurutan. Dalam gambar, komponen seperti *dropout* (Srivastava et al., 2014), *batch normalization* (Ioffe & Szegedy, 2015), dan *layer normalization* (Ba et al., 2016) disembunyikan untuk membuat grafik lebih sederhana. Semua model diatas jatuh kedalam kategori jaringan *encoder-decoder*, dengan *encoder* di kolom kiri dan *decoder* di sisi lainnya. Probabilitas output-nya didapat dari layer proyeksi linear diikuti *sofmax activation* di ujungnya. Semuanya menggunakan mekanisme *attention* di tiap layer *decoder* untuk menghubungkan *encoder* dengan *decoder*, walau masing-masing memiliki spesifikasi yang berbeda. GNMT menggunakan *attention global*, ConvS2S menambah *source word encoding* ke hasilnya, dan *Transformer* menggunakan *multi-head attention*. Residual connection (He et al., 2015) digunakan untuk ketiganya untuk memberikan *gradient flow* dalam jaringan multi-layer. *Positional encoding* digunakan di ConvS2S dan Transformer, namun tidak di GNMT (Stahlberg, 2020).



Gambar 2.3.2 GNMT



Gambar 2.3.3 ConvS2S



Gambar 2.3.4 Transformer

2.3.5 Komponen Dasar NLP dalam NMT

Untuk membangun jembatan pengetahuan yang selaras antar mesin dengan operatornya, dibutuhkan rekayasa menyeluruh pada tiap titik permasalahan. Tiap solusi dari permasalahan tersebut akan dibentuk dan disesuaikan untuk

mencari komposisi yang selaras dalam menghasilkan keluaran yang dapat dimanfaatkan. NMT sebagai dasar fundamental terkini pada NLP terkait penugasan penerjemah mesin tersebut menjadi salah satu buktinya. Terdapat beberapa bidang keilmuan yang digunakan sebagai dasar pembentukan pengetahuan terkait pemrosesan dan penghasilan bahasa dalam NMT. Hal tersebut umumnya berupa perhitungan yang terjadi saat komputasi berjalan didalam sistem, yang membedakannya hanya skalanya yang sudah meluas. (Jurafsky & Martin, 2022) menyimpulkan itu dalam beberapa tema sebagai berikut.

2.3.5.1 Embeddings

Didasarkan dari *representation learning* yang secara otomatis mempelajari representasi dari teks input. Fokus pencariannya dengan mencari cara untuk mempelajari representasi dari input secara *self-supervised,* bukan dengan membuat representasi sendiri melalui *feature engineering* (Bengio et al., 2013).

Didalam NLP digunakan istilah *vector semantic* sebagai standar untuk menyatakan makna dari kata dengan melihat distribusi dalam penggunaannya. Idenya untuk membentuk representasi kata sebagai sebuah point di dalam *multidimensional semantic space* yang berasal dari distribusi kata tetangga. Vektor yang merepresentasikan hal diatas lebih disebut *embeddings*, yang berasal dari sifat matematis untuk memetakan suatu ruang atau struktur ke yang lainnya.

Vektor atau model distribusi dari makna umumnya didasarkan pada matriks jumlah kemunculan, cara untuk mewakili seberapa sering kata-kata berulang kali muncul. Model dasar dari representasi vektor ini berdasar dari perhitungan jumlah pengulangan kata, dimana (Jurafsky & Martin, 2022) menyebutnya sebagai *term-document matrix*.

Dalam matriks *term-document*, setiap baris mewakili kata dalam vocabulary dan setiap kolom mewakili dokumen dari beberapa kumpulan dokumen. Gambar 6.2 menunjukkan pilihan kecil dari matriks *term-document* yang menunjukkan kemunculan empat kata dalam empat drama karya Shakespeare. Setiap sel dalam matriks ini mewakili berapa kali kata tertentu (didefinisikan oleh baris) muncul dalam dokumen tertentu (didefinisikan oleh kolom).

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Gambar 2.3.5 Matriks term-document untuk 4 kata pada karya Shakespeare. Tiap cell terdapat jumlah kata (baris) muncul dan di dalam dokumen (kolom).

Matriks *term-document* dari gambar diatas didefinisikan sebagai model vektor ruang dari pengambilan informasi (Salton, 1971).

Sebuah vektor pada dasarnya hanyalah *list* atau *array* angka. "As You Like It" direpresentasikan sebagai *list* [1.114,36,20] (vektor kolom pertama) dan Julius Caesar direpresentasikan sebagai *list* [7,62,1,2] (vektor kolom ketiga). Vektor ruang adalah kumpulan vektor, yang dicirikan oleh dimensinya. Dalam matriks *term-document* nyata (tidak dengan dimensi 4, seperti contoh), vektor yang mewakili setiap dokumen akan memiliki dimensi |V|, sebagai ukuran vocabulary.

Matriks *term-document* awalnya didefinisikan sebagai sarana untuk menemukan dokumen serupa untuk tugas *information retrieval* pada dokumen. Dua dokumen yang mirip akan cenderung memiliki kata yang mirip, dan jika dua dokumen memiliki kata yang mirip maka vektor kolomnya akan cenderung serupa.

Information retrieval (IR) adalah tugas menemukan dokumen *d* dari dokumen *D* dalam beberapa koleksi yang paling cocok dengan *query q*. Untuk IR kueri akan direpresentasikan dengan vektor, juga dengan panjang |V|, dan diperlukan juga cara untuk membandingkan dua vektor untuk menemukan kemiripannya. (Melakukan IR juga akan membutuhkan cara yang efisien untuk menyimpan dan memanipulasi vektor-vektor ini sebagai *sparse vector*).

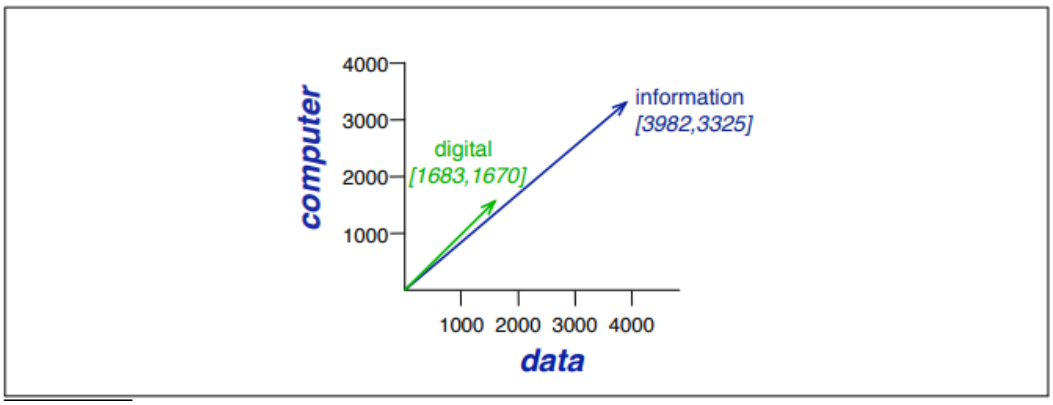
Vector semantics juga dapat digunakan untuk mewakili arti kata-kata. Menggunakan vektor baris daripada vektor kolom, dengan bentuk dimensi yang berbeda. Untuk dokumen yang sama akan memiliki bentuk vektor yang sama, karena dokumen yang sama akan memiliki kata-kata yang sama. Hal tersebut dapat merepresentasikan makna dari sebah kata dengan melihat kecendrungannya muncul di dokumen. Dari contoh gambar xx, empat dimensi vektor untuk *fool*, [36,58,1,4], sesuai dengan empat drama Shakespeare. Jumlah kata dalam empat dimensi yang sama digunakan untuk membentuk vektor untuk 3 kata lainnya: *wit*, [20,15,2,3]; *battle*, [1,0,7,13]; dan *good* [114,80,62,89].

Sebuah alternatif untuk menggunakan matriks *term-document* untuk mewakili kata-kata sebagai vektor dari jumlah dokumen, adalah dengan menggunakan matriks *term-term*, juga disebut matriks *word-word* atau matriks *term-context*, di mana kolom diberi label oleh kata-kata, daripada dokumen. Matriks ini berdimensi |V|×|V| dan setiap sel mencatat berapa kali kata baris (target) dan kata kolom (konteks) muncul bersama dalam beberapa konteks di beberapa korpus pelatihan. Konteksnya bisa berupa dokumen, dalam hal ini sel mewakili berapa kali dua kata muncul dalam dokumen yang sama.

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

Gambar 2.3.6 Vektor Co-occurrence untuk korpus Wikipedia, menampilkan 6 dimensi (dipilih untuk contoh kasus dibawah). Vektor untuk kata ‘digital’ diberi kotak bergaris merah. Sebagai catatan vektor nyata akan memiliki lebih banyak dimensi sehingga akan sangat melebar.

Perhatikan bahwa dua kata *cherry* dan *strawberry* lebih mirip satu sama lain (baik *pie* dan *sugar* cenderung muncul di jendelanya) daripada kata lain seperti *digital;* sebaliknya, *digital* dan *information* lebih mirip satu sama lain daripada, *strawberry*. Gambar 2.3.7 menunjukkan visualisasi spasial terhadap hal tersebut.



Gambar 2.3.7 Visualisasi spasial untuk vektor kata untuk *digital* dan *information*, memperlihatkan hanya dua dimensi, terhadap kata *data* dan *computer*.

Perhatikan bahwa |V|, dimensi dari vektor, umumnya ukuran vocabulary, sering bernilai antara 10.000 dan 50.000 kata (menggunakan derajat kemunculan kata pada korpus training). Namun dikarenakan beberapa kolom (ambil contoh,

aardvark) akan memilki angka yang kosong, tentunya tidak akan banyak membantu. Dikarenakan kolom dengan angka kosong ini terjadi di beberapa lokasi, yang mana akan memberikan kesan vektor yang berjarak, yang mana lebih dikenal sebagai *sparse vector*.

Untuk mengukur kesamaan antara dua kata target v dan w, kita memerlukan metrik yang mengambil dua vektor (dengan dimensi yang sama, baik dengan kata sebagai dimensi, maka dari panjang |V|, atau keduanya dengan dokumen sebagai dimensi sebagai dokumen, dengan panjang |D|) dan memberikan ukuran kesamaan mereka. Sejauh ini metrik kesamaan yang paling umum adalah kosinus sudut antara vektor.

Kosinus, seperti kebanyakan ukuran untuk kesamaan vektor yang digunakan dalam NLP—berdasarkan pada perkalian *dot product*, operator perkalian titik dari aljabar linier, juga disebut *inner product* dalam:

$$\text{dot product}(\mathbf{v},\mathbf{w}) = \mathbf{v}\cdot\mathbf{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \ldots + v_N w_N \tag{2.8}$$

Dot product bertindak sebagai metrik kesamaan karena akan cenderung tinggi hanya ketika dua vektor memiliki nilai tinggi dalam dimensi yang sama. Sebagai alternatifnya, vektor yang memiliki nol dalam dimensi yang berbeda, seperti vektor orthogonal, akan memiliki hasil kali titik 0, yang menunjukkan ketidakmiripan yang kuat.

Dot product dasar ini memiliki masalah sebagai metrik kesamaan, dimana lebih memilih vektor panjang. Panjang vektor didefinisikan sebagai

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2} \tag{2.9}$$

Dot product akan lebih tinggi jika vektor lebih panjang, dengan nilai yang lebih tinggi di setiap dimensi. Kata-kata yang lebih sering muncul akan memiliki vektor yang lebih panjang, karena mereka cenderung muncul bersama dengan lebih banyak kata dan memiliki nilai derajat kemunculan yang lebih tinggi secara keseluruhanmya. *Dot product* dasar ini dengan demikian akan bernilai lebih tinggi untuk kata-kata tersebut. Muncul masalah lain disaat metrik kesamaan tersebut digunakan untuk memberikan informasi seberapa mirip dua kata terlepas dari

frekuensi munculnya.

Dot product dimodifikasi untuk menormalkan panjang vektor dengan membaginya dengan panjang masing-masing dari dua vektor. *Normalized dot product* ini ternyata sama dengan kosinus sudut antara dua vektor, berikut dari definisi produk titik antara dua vektor a dan b:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= |\mathbf{a}||\mathbf{b}| \cos\theta \\ \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} &= \cos\theta \end{aligned} \tag{2.10}$$

Metrik kesamaan kosinus antara dua vektor v dan w dengan demikian dapat dihitung sebagai:

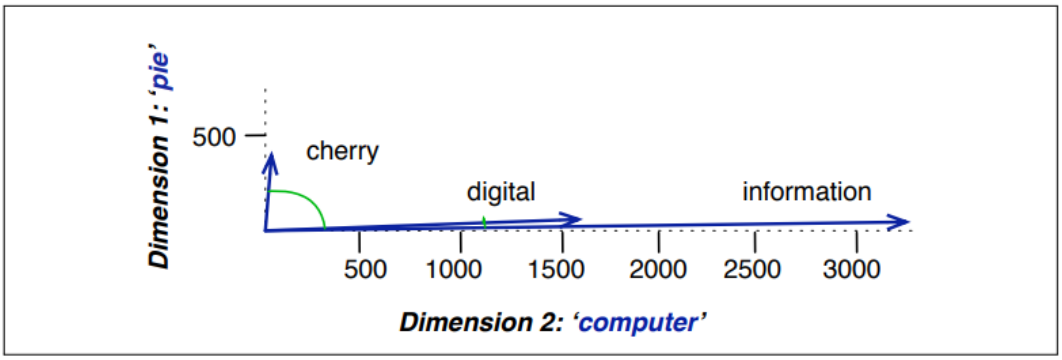
$$\text{cosine}(\mathbf{v}, \mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2} \sqrt{\sum_{i=1}^{N} w_i^2}} \tag{2.11}$$

Untuk pengaplikasian tertentu, vektor butuh untuk dilakukan normalisasi awal, dengan membagi nilainya dengan panjang vektor, membentuk sebuah *unit vector* dengan panjang 1. *Unit vector* tersebut akan dapat dihitung sebagai a dari pembagian |a|. untuk vektor unit, *dot product* akan sama seperti kosinus.

Nilai kosinus memiliki jangkauan dari 1 untuk vektor yang menghadap ke arah yang sama, sementara itu 0 untuk vektor orthogonal, ke -1 untuk kasus yang sama. Dikarenakan nilai frekuensi dasar bukan negative, kosinus untuk vektor tersebut diantara 0-1.

Dengan contoh kasus sebelumnya pada gambar 2.3.6, kosinus dapat dihitung untuk melihat kata *cherry* atau *digital* yang lebih mendekati ke kata *information*.

$$\begin{aligned} \cos(\text{cherry}, \text{information}) &= \frac{442*5 + 8*3982 + 2*3325}{\sqrt{442^2 + 8^2 + 2^2}\sqrt{5^2 + 3982^2 + 3325^2}} = .018 \\ \cos(\text{digital}, \text{information}) &= \frac{5*5 + 1683*3982 + 1670*3325}{\sqrt{5^2 + 1683^2 + 1670^2}\sqrt{5^2 + 3982^2 + 3325^2}} = .996 \end{aligned}$$



Gambar 2.3.8 Demonstrasi grafis untuk *cosine similarity*

Model untuk menghitung kesamaan antara dua kata *x* dan *y* dengan mengambil kosinus dari vektor tf-idf atau PPMI yang mana jika nilai kosinus tinggi, maka derajat kesamaan tinggi. Keseluruhan model ini kadang-kadang disebut sebagai model *tf-idf* (Luhn, 1957; Sparck Jones, 1972) atau model *PPMI* (Church & Hanks, 1989; Dagan et al., 1993; Niwa & Nitta, 1994). pembobotan *tf-idf* yang menimbang setiap sel dengan frekuensi term dan frekuensi dokumen terbalik, dan PPMI (*Positive Pointwise Mutual Information*), yang paling umum untuk matriks kata-konteks atau word-word dengan menentukan asosiasi terhadap kedua kata yang saling berhubungan.

Selain representasi model vektor sparse diatas, tersebutlah *embeddings* sebagai vektor dengan *short-density*. Tidak seperti vektor sparse, *embeddings* berukuran pendek, dengan jumlah dimensi *d* berkisar antara 50-1000, daripada ukuran vocabulary yang jauh lebih besar |V| atau jumlah dokumen *D* sebelumnya. Dimensi *d* ini tidak memiliki interpretasi yang jelas. Dan vektornya sangat padat karena sebagian besar jumlah nol atau fungsi penghitungan, nilainya akan menjadi bilangan bernilai nyata yang bisa negatif.

Untuk menggambarkan representasi vektor *dense* dan komputasinya, digunakan algoritma dari *word2vec* (Mikolov et al., 2013b, 2013a) dengan algoritma *skip-gram with negative sampling* atau SGNS sebagai dasar pengenalannya. *Embeddings* dari Word2vec adalah *embeddings* statis, artinya metode ini mempelajari satu embeddings tetap untuk setiap kata dalam vocabulary atau vocabulary.

Intuisi dari word2vec adalah bahwa daripada menghitung seberapa sering setiap kata *w* muncul berdekatan, *classifier* akan dilatih pada tugas *binary*

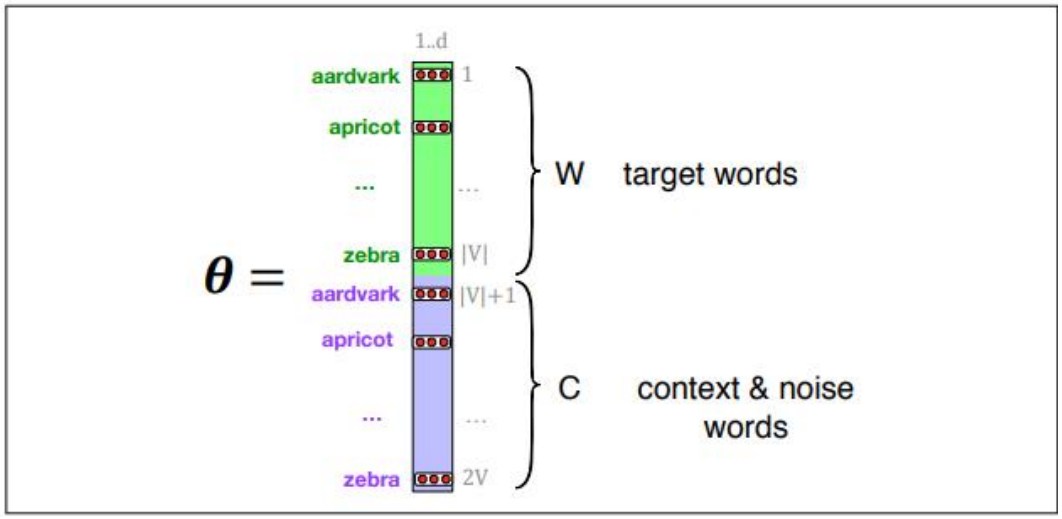
prediction, lalu ambil bobot pengklasifikasi yang dipelajari sebagai *embeddings* kata. Pengklasifikasi tersebut dapat menggunakan teks berjalan sebagai data pelatihan supervised secara implisit. Metode ini, sering disebut *self-supervision*, menghindari kebutuhan akan segala jenis sinyal pengawasan yang diberi label tangan. Ide ini pertama kali diusulkan dalam tugas pemodelan bahasa saraf, ketika (Bengio et al., 2003) dan (Collobert et al., 2011) menunjukkan bahwa model bahasa saraf (jaringan saraf yang belajar memprediksi kata berikutnya dari kata-kata sebelumnya) hanya dapat menggunakan kata berikutnya dalam menjalankan teks sebagai sinyal pengawasan, dan dapat digunakan untuk mempelajari representasi embedding untuk setiap kata. sebagai bagian dari melakukan tugas prediksi ini.

$$P(+|w, c_{1:L}) = \prod_{i=1}^{L} \sigma(\mathbf{c_i} \cdot \mathbf{w}) \tag{2.12}$$

$$\log P(+|w, c_{1:L}) = \sum_{i=1}^{L} \log \sigma(\mathbf{c_i} \cdot \mathbf{w}) \tag{2.13}$$

Algoritma skip-gram melatih pengklasifikasi probabilistik yang, diberi kata target uji *w* dan jendela konteksnya dari kata *L* $c_{1:L}$, memberikan probabilitas berdasarkan seberapa mirip jendela konteks ini dengan kata target. Probabilitas didasarkan pada penerapan fungsi logistik (sigmoid) ke dot product dari embeddings kata target dengan setiap kata konteks. Untuk menghitung probabilitas ini, kita hanya perlu embeddings untuk setiap kata target dan kata konteks dalam vocabulary.

Skip-gram menggunakan stochastic gradient descent untuk melatih classifier, dengan mempelajari embeddings yang memiliki nilai dot product tinggi dengan embeddings dari kata-kata yang berada bersebelahan dan low dot product dengan noise words.



Gambar 2.3.9 Embeddings yang dipelajari oleh model skipgram. Algoritma menyimpan dua embedding untuk setiap kata, embeddings target (kadang-kadang disebut embedding input) dan embedding context (kadang-kadang disebut embedding output). Parameter yang dipelajari algoritma adalah matriks 2|V| vektor, masing-masing dimensi *d*, dibentuk dengan menggabungkan dua matriks, embeddings target **W** dan embeddings konteks+noise **C**

2.3.5.2 Neural Networks

Neural network atau Jaringan Saraf Tiruan adalah alat komputasi yang fundamental untuk pemrosesan bahasa, dan yang tertua (MCCulloch and Pits, 1943). *Neural network* modern merupakan network yang berisi kelompok kecil unit komputasi, tiap bagiannya mempunyai vektor dari nilai input dan memproduksi nilai output tunggal. Penggunaan modern neural network biasanya disebut *deep learning*, karena biasanya memiliki *layer* yang bertingkat sehingga terkesan dalam *(deep*).

Neural network mendapat bagian yang sama fungsi matematis sebagai *logistic regression*. Namun *neural network* mempunyai *classifier* yang lebih baik dari *logistic regression* tersebut, karena penggunaannya yang minimal (karena ada hidden layer) untuk mempelajari fungsi apapun.

Bagian terkecil *neural network* disebut sebagai sebuah unit pemrosesan tunggal (*neural units*), yang bertugas mengambil sekelompok nilai angka asli sebagai input, melakukan komputasi terhadapnya, dan memperoleh output.

Dalam aljabar linier, bagian ini dapat digambarkan sebagai notasi vektor yang mana dasarnya hanya berupa *list* atau *array* dari bilangan. Disebutkan z sebagai *weight vector* w, *scalar bias* b, dan input vektor x, seperti yang dijelaskan

di persamaan 2.8

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{2.8}$$

Selanjutnya, dengan mengganti penggunaan z, fungsi linear x, sebagai output, neural unit menggunakan fungsi non-linear f ke z. output dari hasil ini lebih umum dinamakan activation value untuk unit, a. Dalam unit tunggal dengan final output y, penggambarannya dapat dilihat di persamaan 2.9.

$$y = a = f(z) \tag{2.9}$$

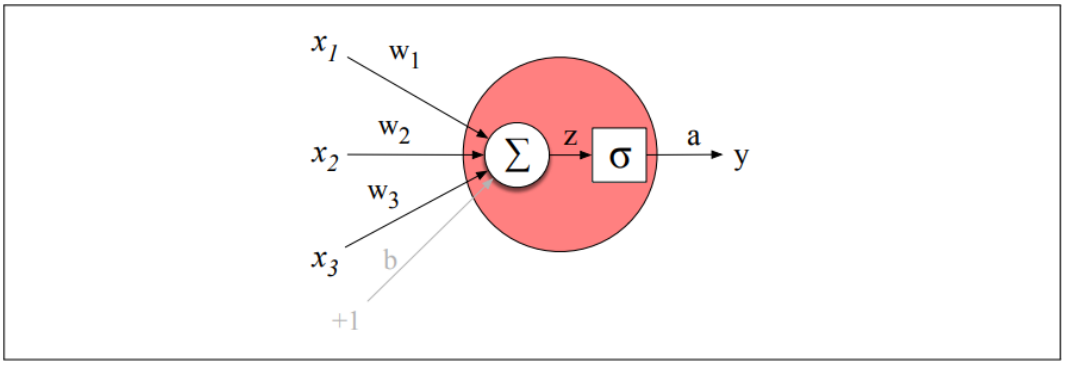
Fungsi non-linear yang umum digunakan sebagai $f()$, antara lain *sigmoid*, *tanh*, dan *rectified linear unit* atau ReLU. Yang mana jika dalam kasus penggunaan *sigmoid* akan membentuk persamaan 2.10.

$$y = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.10}$$

Dengan itu, jika semua bagian persamaan digabungkan, menjadi salah satu model yang bekerja dalam *neural unit*.

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))} \tag{2.11}$$

Gambar xx menunjukkan skema akhir dari dasar sebuah *neural unit*. Di contoh tersebut digambarkan unit menerima 3 nilai input x1, x2, dan x3, dan melakukan perhitungan *weighted sum*, mengalikan tiap nilai dengan bobotnya (w1, w2, dan w3, berurutan), memberikannya *bias* b, dan menyampaikannya ke penjumlahan hasil melalui fungsi *sigmoid* untuk mendapat keluaran dalam bentuk angka antara 0 dan 1 pada a .



Gambar 2.3.10 Sebuah unit saraf, mengambil 3 input x_1, x_2, dan x_3 (dan bias *b* yang dinyatakan sebagai bobot untuk input yang dijepit pada +1) dan menghasilkan output *y*.

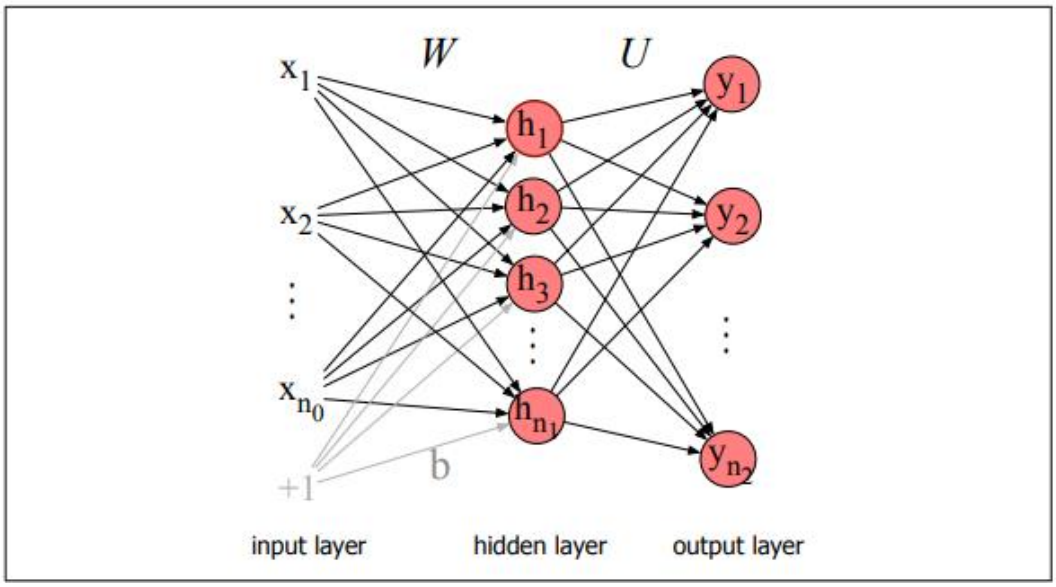
Beberapa jenis *activation function* memiliki cirinya masing-masing, yang membuat mereka berguna di kondisi tertentu dalam penggunaannya di aplikasi bahasa atau arsitektur jaringan berbeda. Sebagai contoh, fungsi *tanh* memiliki fungsi untuk diferensiasi yang halus dan pemetaan nilai *outlier* menuju rata-rata. Fungsi *ReLU*, memfokuskan hasil untuk mendekati hasil yang linear. Di fungsi *sigmoid* atau *tanh*, nilai *z* yang terlalu tinggi akan menghasilkan nilai dari *y* yang jenuh (*saturated*), dengan nilai yang terlalu mendekati angka 1, dan turunan yang mendekati angka 0. Turunan 0 akan mengkakibatkan masalah untuk pembelajaran, karena training jaringan dengan menyebarkan sinyal error kebelakang (back propagation), *multiplying gradient* (turunan sebagian) dari tiap *layer* jaringan yang memiliki nilai *gradient* mendekati 0 akan menyebabkan sinyal error untuk mengecil sampai titik terkecil untuk digunakan saat training. Umumnya hal ini disebut masalah *vanishing gradient*, yang mana fungsi *rectifier* tidak memilikinya, dikarenakan turunan dari ReLU untuk hasil yang tinggi dari *z* adalah 1 atau mendekati 0.

2.3.5.2.1 Feedforward Neural Networks

Feedforward network adalah jaringan multilayer sederhana, yang mana tiap unit akan terhubung dengan tidak ada perputaran (sekali jalan). Output dari unit di tiap layer disampaikan ke unit yang berada di layer atas berikutnya, dan tidak ada output yang dibawa kembali ke layer bawah.

Feedforward network pada umumnya memiliki tiga jenis node, input unit, hidden unit, dan output unit. Input layer x adalah vektor dari nilai skalar. Adapun itu seperti pada contoh gambar 2.3.11.

Inti dari neural network adalah *hidden layer h* yang terbentuk dari *hidden unit* h_i, tiap bagian neural unit akan menerima weighted sum dari inputnya dan mengaplikasikan fungsi non-linier. Pada arsitektur standarnya, tiap layer akan terhubung penuh (*fully-connected*), dengan maksud tiap unit di tiap layer-nya mengambil output dari semua unit yang berada di layer sebelumnya sebagai input. Bagian tersebut, terutama yang berasal dari dua layer berdekatan, akan memiliki hubungan dan memungkinkan tiap hidden unit untuk menjumlahkan hasil keseluruhan dari input unit.



Gambar 2.3.11 *Feedforward network* sederhana dengan 2 layer, satu sebagai *hidden layer*, satunya sebagai *output layer*, dan *input layer* (input layer tidak dihitung ketika perhitungan layer terjadi)

Ingat kembali bahwa satu hidden unit memiliki parameter vektor bobot dan bias. Representasi parameter untuk seluruh hidden layer diperoleh dengan menggabungkan vektor bobot dan bias untuk setiap unit-i ke dalam matriks bobot tunggal **W** dan vektor bias tunggal *b* untuk seluruh layer. Setiap elemen $\mathbf{W}_{ji}$ dari matriks bobot **W** mewakili bobot koneksi dari unit input ke-*i* x_i ke hidden unit ke-*j* h_j.

Keuntungan menggunakan matriks tunggal **W** untuk bobot seluruh lapisan layer adalah perhitungan layer tersembunyi untuk jaringan feedforward dapat

dilakukan dengan sangat efisien hanya dengan operasi matriks sederhana. Faktanya, perhitungan hanya memiliki tiga langkah: mengalikan matriks bobot dengan vektor input **x**, menambahkan vektor bias **b**, dan menerapkan fungsi aktivasi *g* (seperti fungsi aktivasi sigmoid, tanh, atau ReLU yang didefinisikan di atas).

Output dari *hidden layer*, vektor *h*, jika dihitung dengan menggunakan fungsi sigmoid σ sebagai fungsi aktivasi, maka akan seperti berikut.

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.12}$$

Pengalian matriks di persamaan 2.12 akan menghitung nilai dari tiap h_j sebagai $\sigma\left(\sum_{i=1}^{n_0} W_{jixi} + b_j\right)$.

Nilai **h** akan membentuk representasi dari input, dimana tugas output layer untuk mengambil representasi terbaru dari **h** dan menghitung keluaran final. Keluarannya boleh berupa bilangan asli, tetapi dalam banyak kasus tujuan dari jaringan tersebut adalah untuk membuat semacam keputusan akan klasifikasi.

Pada tugas binary seperti klasifikasi sentimen, kemungkinan hanya akan memiliki satu node keluaran, dan nilai skalarnya *y* adalah probabilitas sentimen positif versus negatif. Jika melakukan klasifikasi multinomial, seperti menetapkan tag *part-of-speech*, kemungkinan akan memiliki satu node output untuk setiap *part-of-speech* potensial, yang nilai outputnya adalah probabilitas *part-of-speech* itu, dan nilainya dari semua node keluaran harus berjumlah satu. Lapisan keluaran dengan demikian adalah vektor *y* yang memberikan distribusi probabilitas di seluruh node keluaran.

Komputasi yang terjadi sebelum menemukan *node output final* akan menggunakan representasi **h** tersebut, dimana bentuknya dalam vektor. Persamaannya didasari *weight matrix* **U** dikalikan *input vector* **h** untuk menghasilkan output penghubung **z**.

$$\mathbf{z} = \mathbf{U}\mathbf{h} \tag{2.13}$$

Di akhir node, output akan dihasilkan, namun **z** tidak dapat menjadi hasil tersebut. Untuk mengkondisikannya, dibutuhkan fungsi normalisasi yang mengubah bentuk vektor tersebut menjadi bilangan asli (antara 1 dan 0). Tersebutlah fungsi *softmax,* dengan persamaan.

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d}\exp(\mathbf{z}_j)} \quad 1 \le i \le d \tag{2.14}$$

Neural network berlaku seperti *multinomial logistic regression*, dengan banyak layer, yang masing-masing memiliki fungsi aktivasi tertentu, dan dibandingkan dengan membentuk *feature* dari *feature template*, layer pertama dari jaringan memunculkan *feature representation* dengan sendirinya.

2.3.5.2.2 Feedforward Neural Language Modeling

Feedforward Neural Language Modeling dikenalkan pertama kali oleh (Bengio et al, 2003) sebagai metode *language modeling* yang dibutuhkan untuk memprediksi keluaran kata selanjutnya dari konteks kata sebelumnya. Menjadi dasar pertama dari *language model* atau LM berbasis *neural network*, *feedforward neural* LM ini mendapat keuntungan dari mesin dan skala tugas yang besar, beda dengan n-grams yang lebih cocok untuk kondisi sebaliknya.

Feedforward neural LM mengambil input *t* pada satu waktu, representasi dari sejumlah angka dari kata sebelumnya (w_{t-1}, w_{t-2}, dsb), dan menghasilkan sebuah probabilitas distribusi terhadap kemungkinan kata berikutnya. *Feedforward neural* LM memperkirakan probabilitas dari sebuah kata yang diberikan dari seluruh konteks awal $P(w_t \mid w_{1:t-1})$ dengan perkiraan yang berdasar terhadap *N-1* kata sebelumnya, seperti di persamaan berikut.

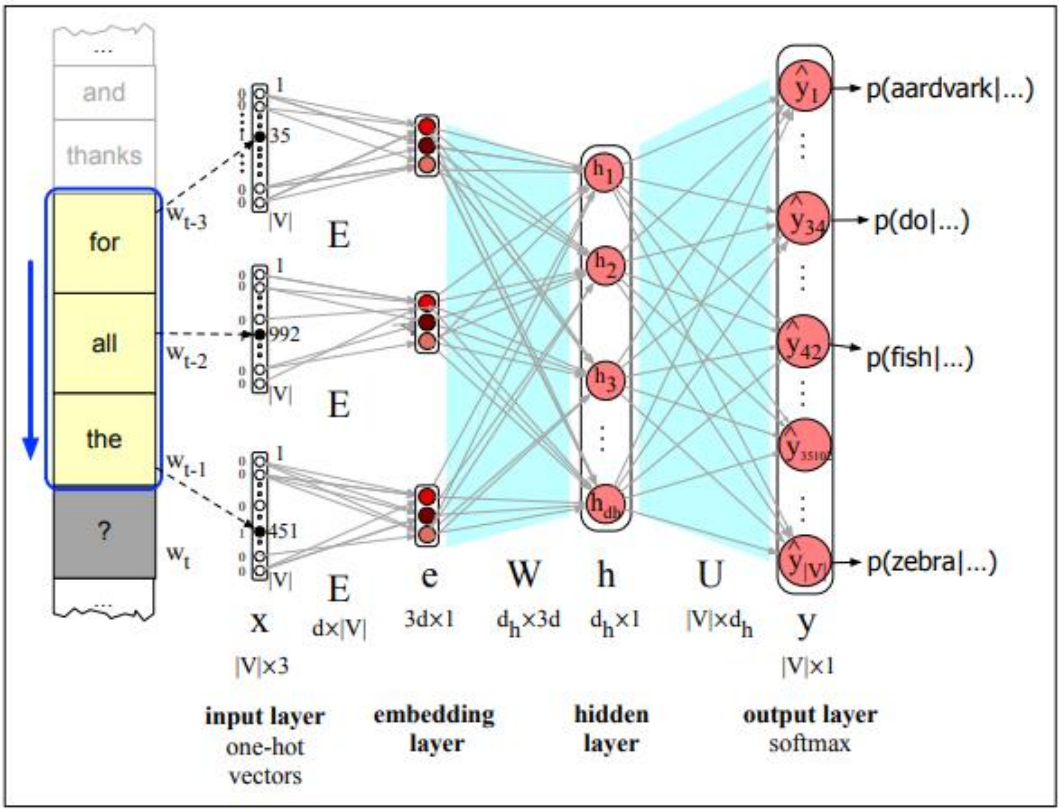
$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1}) \tag{2.15}$$

Forward pass akan dilakukan ke jaringan untuk memproduksi sebuah probabilitas distribusi terhadap kemungkinan keluaran, dalam kasus ini kata selanjutnya.

Pada tiap *timestep t* jaringan akan mengolah *d-dimensional embedding* untuk tiap konteks kata (dengan mengalikan sebuah *one-hot vector* dengan matriks embedding **E**), dan menyatukan 3 penghasil embedding untuk mendapatkan *embedding layer* **e**. *Embedding vector* **e** akan dikalikan dengan *weight matrix* **W** dan fungsi aktivasi akan diaplikasikan untuk memproduksi *hidden layer* **h**, yang mana akan dikalikan oleh *weight matrix* **U**. Terakhir, *softmax output layer* akan memprediksi di tiap *node i* dengan kemungkinan kata selanjutnya w_t merupakan *vocabulary word* V_i. Adapun persamaannya sebagai berikut.

$$\begin{aligned} \mathbf{e} &= [\mathbf{E}\mathbf{x}_{t-3}; \mathbf{E}\mathbf{x}_{t-2}; \mathbf{E}\mathbf{x}_{t-1}] \\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{e} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{2.16}$$

Dan jika digambarkan akan terlihat seperti ini.



Gambar 2.3.12 *Forward inference* dalam sebuah *feedforward neural language model*.

2.3.5.2.3 Training Neural Network

Feedforward neural network adalah instance dari supervised machine learning pada yang mana diketahui output *y* dari tiap observasi **x**. Apa yang sistem hasilkan dari perasamaan 2.16, adalah ŷ, sistem memperkirakan nilai asli *y*. Tujuan utama dari prosedur training adalah untuk mempelajarai parameter $\mathbf{W}^{[i]}$ dan $\mathbf{b}^{[i]}$ pada tiap layer *i* yang membuat ŷ untuk tiap observasi training mendekati hasil *y* asli.

Pertama, dibutuhkan sebuah *loss function* yang mampu memodelkan jarak terhadap output dari sistem dan output terbaik, dan umum untuk menggunakan *loss function* yang digunakan untuk logistic regression, *cross-entropy loss*.

Kedua, untuk menemukan parameter yang dapat mengecilkan *loss function* ini. Digunakanlah algoritma optimasi *gradient descent*.

Ketiga, *gradient descent* membutuhkan pengetahuan terkait *gradient* dari *loss function*, vektor yang terdapat *partial derivative* dari *loss function* dari tiap parameter terkait. Dalam *logistic regression*, untuk tiap observasi dapat dihitung langsung turunan dari *loss function*, yaitu terhadap *w* atau *b* individual. Tetapi untuk *neural network*, dengan ber-miliaran parameter di banyak layer, akan sangat sulit untuk melihat bagaimana cara perhitungan terhadap *partial derivative* dari *weight* di layer 1 ketika *loss* terikat pada sebagian banyak layer lainnya. Untuk membagi bagian *loss* terhadap semua *intermediate layer* dibutuhkan algoritma *error backpropagation* atau *backward differentiation*.

2.3.5.2.3.1 Loss Function

Cross-entropy loss yang digunakan dalam neural network mrtupakan salah satu yang ada di *logistic regression*. Jika *neural network* digunakan sebagai *binary classifier*, dengan *sigmoid* di layer terakhirnya, *loss function* akan sama dengan *logistic regression loss*.

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \quad (2.17)$$

Jika menggunakan jaringan untuk mengklasifikasikan kedalam 3 atau lebih kelas, *loss function* akan sama dengan *loss* untuk *multinomial regression*. Pertama, jika didapat dua kelas maka $\mathbf{y}$ dan $\hat{\mathbf{y}}$ harus direpresentasikan kedalam bentuk vektor. Dalam *hard classification*, hanya satu kelas saja yang akan benar. Tersebut *true label* $\mathbf{y}$ kemudian sebuah vektor dengan *K* elemen, yang menyesuaikan dengan kelasnya, dengan $\mathbf{y_c}$=1 jika kelas yang benar adalah *c*, dengan elemen lainnya dari $\mathbf{y}$ menjadi 0. Keadaan vektor seperti ini, yang memiliki suatu nilai yang sama dengan 1 sedang yang lainnya 0, disebut sebagai *one-hot vector*. Dari klasifikasi tersebut akan dihasilkan estimasi vektor dengan *K* elemen $\hat{\mathbf{y}}$, tiap elemen $\hat{\mathbf{y}}_\mathbf{k}$ dari yang menunjukkan estimasi probabilitas $p(\mathbf{y}_k = 1|x)$.

Loss function untuk sebuah contoh x adalah penjumlahan negatif dari *logs output* kelas *K*, yang tiap bagiannya dibobotkan terhadap probabilitasnya y_k:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{2.18}$$

Persamaan diatas dapat disederhanakan; fungsi akan ditulis ulang dengan menggunakan function 1{} yang mengevaluasi ke angka 1 jika kondisi dalam bracket benar dan ke 0 sebaliknya. Hal tersebut memperjelas ketentuan dalam persamaan 2.18 untuk menjadi 0 kecuali untuk ketentuan yang terkait terhadap kelas sebenarnya untuk $\mathbf{y}_k = 1$;

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbb{1}\{\mathbf{y}_k = 1\} \log \hat{\mathbf{y}}_k \tag{2.19}$$

Dalam penjelasan lainnya, *cross-entropy loss* sederhanyanya adalan *log negative* dari probabilitas output yang terhubung ke kelas sebenarnya. Hal ini umum disebut sebagai *negative log likelihood loss*:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \hat{\mathbf{y}}_c \quad \text{(where } c \text{ is the correct class)} \tag{2.20}$$

Jika digabungkan dengan rumus sofmax di persamaan xx, dan dengan K yang berupa jumlah dari kelasnya:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \quad \text{(where } c \text{ is the correct class)} \tag{2.21}$$

2.3.5.2.3.2 Menghitung Gradient

Untuk menghitung *gradient* dari *loss function* dibutuhkan *partial derivative* dari fungsi loss yang berhubungan ke tiap parameter. Untuk sebuah *network* dengan satu layer *weight* dan *output sigmoid* (yang berasal dari *logistic regression*), sederhananya dapat digunakan turunan dari *loss* saat penggunaan di *logistic regression* di persamaan 2.22 (dan yang diturunkan):

$$\begin{aligned} \frac{\partial L_{CE}(\hat{\mathbf{y}}, \mathbf{y})}{\partial w_j} &= (\hat{y} - y)\mathbf{x}_j \\ &= (\sigma(\mathbf{w} \cdot \mathbf{x} + b) - y)\mathbf{x}_j \end{aligned} \tag{2.22}$$

Atau untuk sebuah network dengan satu layer *weight* dan *output softmax* (*multinomial logistic regression*), dapat digunakan turunan *softmax loss* yang ditunjukkan dengan *weight* $\mathbf{w}_k$ dan input $\mathbf{x}_i$ tertentu.

$$\begin{aligned}\frac{\partial L_{\text{CE}}(\hat{\mathbf{y}},\mathbf{y})}{\partial \mathbf{w}_{k,i}} &= -(\mathbf{y}_k - \hat{\mathbf{y}}_k)\mathbf{x}_i \\ &= -(\mathbf{y}_k - p(\mathbf{y}_k = 1|\mathbf{x}))\mathbf{x}_i \\ &= -\left(\mathbf{y}_k - \frac{\exp(\mathbf{w_k}\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)}\right)\mathbf{x}_i\end{aligned} \tag{2.23}$$

Akan tetapi turunan diatas hanya akan memberikan pembaruan yang tepat untuk satu layer *weight* saja dan itu di akhir. Untuk *deep network*, menghitung *gradient* untuk tiap *weight* bisa sangat kompleks, karena perhitungan turunan dari tiap parameter terhubung dengan yang muncul terbelakang dari keseluruhan layer awal network, meskipun *loss* dihitung saat di ujung akhir network.

Sebagai solusi untuk ini tersebutlah algoritma *error backpropagation* atau *backprop* (Rumelhart et al., 1986). Walau *backprop* dirancang khusus untuk *neural network*, nyatanya bersifat sama dengan prosedur biasa yang dinamakan *backward differentiation*, yang dipercaya mengandalkan komputasi grafik.

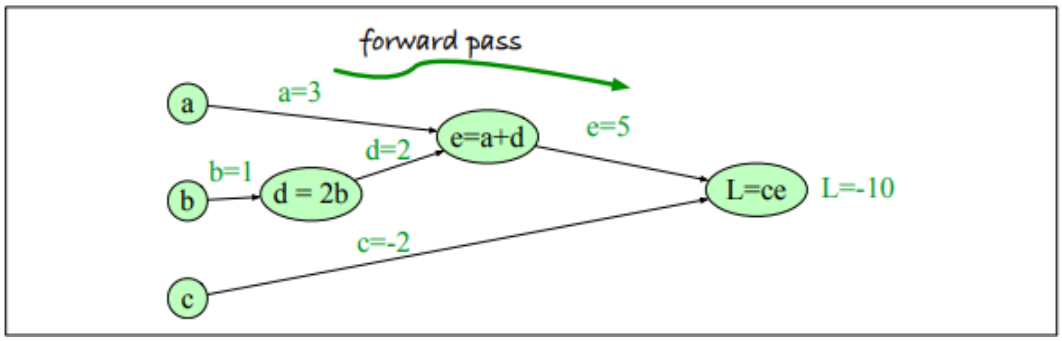
2.3.5.2.3.3 Computation Graphs

Komputasi grafis adalah sebuah representasi dari proses perhitungan sebuah ekspresi matematis, yang mana prosesnya akan dipecah menjadi beberapa operasi, yang tiap bagiannya dimodelkan sebagai sebuah node di grafik.

Misalkan perhitungan dari fungsi $L(a,b,c) = c(a+2b)$. Jika masing-masing dari komponen dari operasi pertambahan dan perkalian jelas, dan tambah nama (d dan e) untuk output *intermediate*, hasil dari komputasinya jika dikelompokkan adalah sebagai berikut.

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned} \tag{2.24}$$

Grafik akan dapat direpresentasikan, dengan node untuk tiap operasi, dan tepian terarah yang menunjukkan output dari tiap operasi sebagai input untuk menuju ke yang berikutnya. Penggunaan komputasi grafik sederhananya untuk menghitung nilai dari fungsi dengan bermacam input.



Gambar 2.3.13 Computation graph untuk fungsi $L(a,b,c) = c(a+2b)$, dengan nilai untuk node inputnya, menunjukkan komputasi *forward pass* dari L

Pada gambar diatas, ditetapkan input $a = 3$, $b = 1$, $c = -2$, dan hasil dapat dilihat sebagaimana *forward pass* digunakan untuk menghitung hasil dari $L(3,1,-2) = -10$. Dalam *forward pass* dari komputasi grafik, diaplikasikan tiap operasi dari kiri ke kanan, membagi output dari tiap perhitungan sebagai input untuk node berikutnya.

2.3.5.2.3.4 Backward differentiation pada komputasi grafis

Hal penting dari komputasi grafis dating dari backward pass, yang mana digunakan untuk menghitung turunan saat pembaruan weight. Dicontohkan untuk melakukan perhitungan turunan terhadap fungsi output L yang terhubung di tiap variable input $(\frac{\partial L}{\partial a}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial c})$. Turunan $\frac{\partial L}{\partial a}$, akan memberikan keterangan seberapa kecil efek perubahan a ke L.

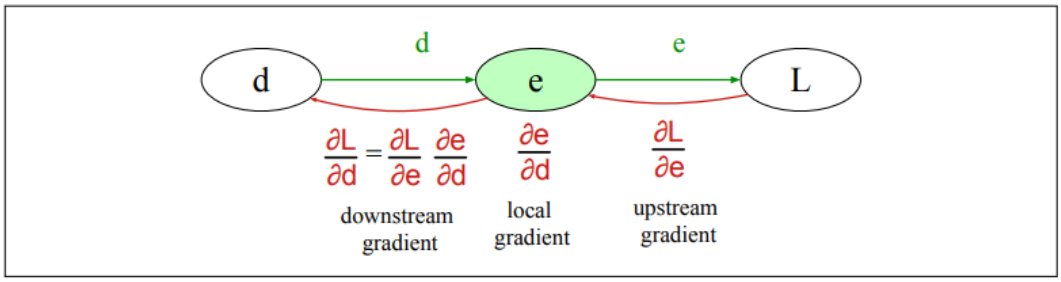
Backward differentiation menggunakan *chain rule* pada kalkulus. Anggap ada suatu kasus perhitungan turunan dari fungsi komposit $f(x) = u(v(x))$. Turunan dari $f(x)$ adalah turunan dari $u(x)$ terhadap ke $v(x)$ yang dikalikan turunan dari $v(x)$ terhadap x.

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{2.25}$$

Chain rule akan melebar untuk dua fungsi. Jika perhitungan dari turunan sebauh fungsi komposit $f(x) = u(v(w(x)))$, turunan $f(x)$ menjadi

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx} \tag{2.26}$$

Intuisi dari *backward differentiation* adalah untuk memberikan *gradient* kembali dari final node ke semua node dalam grafik.



Gambar 2.3.14 *Backward computation* pada satu node *e*.

Gambar diatas menunjukkan sebagian dari *backward computation* pada satu node *e*. Tiap node mengambil *upstream gradient* yang disampaikan dari *node parent* di kanan, dan untuk tiap input-nya akan menghitung *gradient local* (*gradient* dari output yang terhubung dengan input), dan menggunakan *chain rule* untuk mengalikan keduanya untuk menghitung sebuah *downstream gradient* untuk diberikan ke node awal pertama.

Ketiga turunan tersebut dapat dihitung. Karena berada di komputasi grafis $L = ce$, dapat dihitung secara langsung turunan $\frac{\partial L}{\partial c}$.

$$\frac{\partial L}{\partial c} = e \tag{2.27}$$

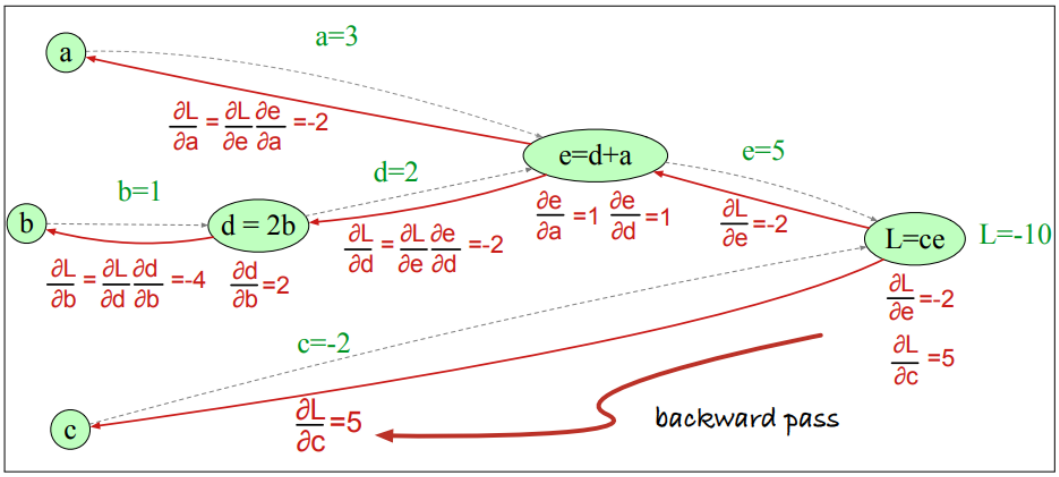
Untuk dua lainnya, akan digunakan chain rule.

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned} \tag{2.28}$$

Persamaan 2.27 dan 2.28 membutuhkan lima turunan *intermediate*: $\frac{\partial L}{\partial e}, \frac{\partial L}{\partial c}, \frac{\partial e}{\partial a}, \frac{\partial e}{\partial d}$, dan $\frac{\partial d}{\partial b}$, yang mana (menggunakan fakta turunan dari penjumlahan yang merupakan penjumlahan dari turunan):

$$\begin{aligned} L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\ e = a + d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\ d = 2b &: \quad \frac{\partial d}{\partial b} = 2 \end{aligned} \tag{2.29}$$

Pada *backward pass*, dihitung tiap parsial tersebut dengan tiap sisi dari grafik dari kanan ke kiri, menggunakan *chain rule* diatas. Dilanjutkan dengan menghitung *downstream gradient* dari node *L*, yang mana $\frac{\partial L}{\partial e}, \frac{\partial L}{\partial c}$. Untuk node *e*, akan dikalikan *upstream gradient* $\frac{\partial L}{\partial e}$ dengan *local gradient* (*gradient* dari output yang terhubung ke input), $\frac{\partial e}{\partial d}$ untuk mendapat output yang dikirim kembali ke node d: $\frac{\partial L}{\partial d}$. Berikutnya terus menerus, sampai semua input variabel grafik tercatat semua. *Forward pass* dengan mudah akan mempersiapkan hitungan nilai dari *forward intermediate variable* yang dibutuhkan (seperti *d* dan *e*) untuk menghitung turunan tersebut.



Gambar 2.3.15 *Backward pass* untuk *computation graph* fungsi $L(a,b,c) = c(a+2b)$.

2.3.5.2.3.5 Backward differentiation untuk neural network

Komputasi grafis tentunya akan makin kompleks untuk *neural network* sebenarnya. Berikut disediakan contoh komputasi grafik untuk neural network dengan 2-layer dengan $n_0 = 2$, $n_1 = 2$, dan $n_2 = 1$, dengan anggapan menggunakan *binary classification* dan *sigmoid output unit* untuk kesederhanaannya. Fungsi dari perhitungan komputasi grafik tersebut sebagai berikut:

$$\begin{aligned} \mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\ \mathbf{a}^{[1]} &= \text{ReLU}(\mathbf{z}^{[1]}) \\ z^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned} \tag{2.30}$$

Untuk *backward pass* dibutuhkan juga perhitungan *loss L*. fungsi *loss* untuk *binary sigmoid* output dibentuk sebagai berikut:

$$L_{CE}(\hat{y},y) = -[y\log\hat{y}+(1-y)\log(1-\hat{y})] \tag{2.31}$$

Untuk output $\hat{y} = a^{[2]}$, dapat ditulis sebagai berikut

$$L_{CE}(a^{[2]},y) = -\left[y\log a^{[2]}+(1-y)\log(1-a^{[2]})\right] \tag{2.32}$$

Bobot yang butuh diperbarui (bagian yang dibutuhkan untuk mengetahui turunan parsial dari *loss function*) diperlihatkan dengan warna teal. Sebagai kondisi untuk melakukan *backward pass*, perlu diketahui turunan dari semua fungsi dalam grafik. Digunakanlah turunan *sigmoid* σ:

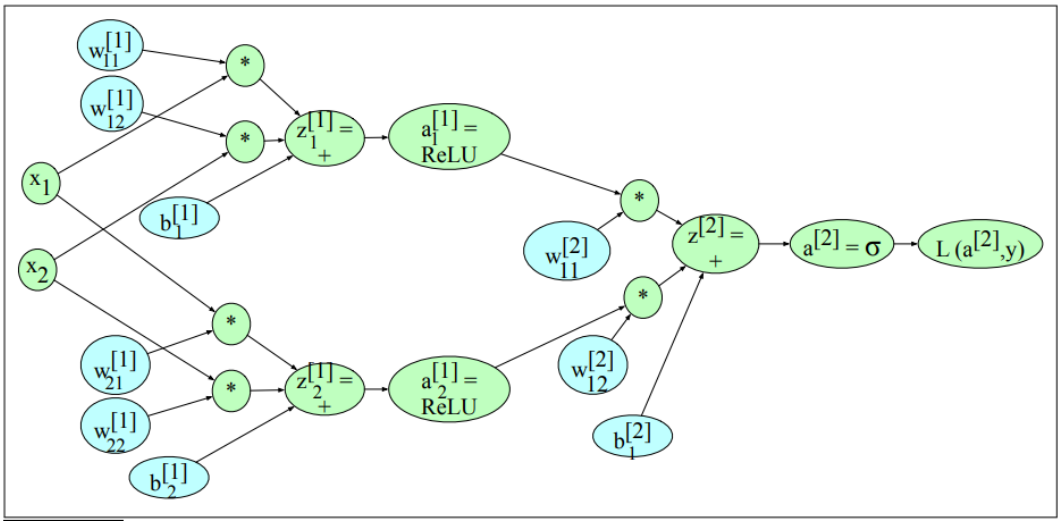
$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{2.33}$$

Juga dibutuhkan turunan dari tiap *activation function* lainnya. Turunan *tanh*.

$$\frac{d\tanh(z)}{dz} = 1-\tanh^2(z) \tag{2.34}$$

Sementara itu turunan untuk *ReLU* adalah

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \;\; z<0 \\ 1 & for \;\; z\geq 0 \end{cases} \tag{2.35}$$



Gambar 2.3.16 Contoh *computation graph* untuk 2-layer *neural network* sederhana (= 1 *hidden layer*) dengan dua *input units* dan 2 *hidden units*.

2.3.5.2.3.6 Penjelasan Tambahan Untuk Proses Learning

Optimasi dalam *neural network* adalah masalah optimasi *non-convex*, lebih kompleks daripada *regresi logistik*, dan untuk itu dan alasan lain ada banyak praktik terbaik untuk pembelajaran yang sukses.

Untuk *logistic regression*, dapat menginisialisasi *gradient descent* dengan semua bobot dan bias yang memiliki nilai 0. Sebaliknya, dalam jaringan saraf, perlu menginisialisasi bobot dengan angka acak kecil. Ini juga membantu untuk menormalkan nilai input agar memiliki 0 *mean* dan varians unit.

Berbagai bentuk regularisasi digunakan untuk mencegah *overfitting*. Salah satu yang paling penting adalah *dropout*: secara acak menjatuhkan beberapa unit dan koneksinya dari jaringan selama pelatihan (Hinton et al., 2012; Srivastava et al., 2014). *Tuning* dari *hyperparameters* juga penting. Parameter jaringan saraf adalah bobot *W* dan bias *b*; yang dipelajari dengan *gradient descent*. *Hyperparameter* adalah hal-hal yang dipilih oleh perancang algoritma; nilai-nilai optimal disetel pada *devset* daripada oleh pembelajaran *gradient descent* pada *training set*. Hyperparameter termasuk kecepatan belajar η, ukuran *mini-batch*, arsitektur model (jumlah layer, jumlah *hidden node* per *layer*, pilihan fungsi aktivasi), cara untuk *regularize*, dan sebagainya. *Gradient descent* sendiri juga memiliki banyak varian arsitektur seperti *Adam* (Kingma & Ba, 2014), *Adadelta* (Zeiler, 2012), *RmsProp* (Tieleman et al., 2012).

Pada akhirnya, sebagian besar jaringan saraf modern dibangun menggunakan formalisme komputasi grafis yang membuatnya mudah dan alami untuk melakukan komputasi *gradient* dan paralelisasi pada GPU berbasis vektor (*Graphic Processing Units*). *PyTorch* (Paszke et al., 2019) dan *TensorFlow* (Abadi et al., 2016) adalah dua yang paling populer.

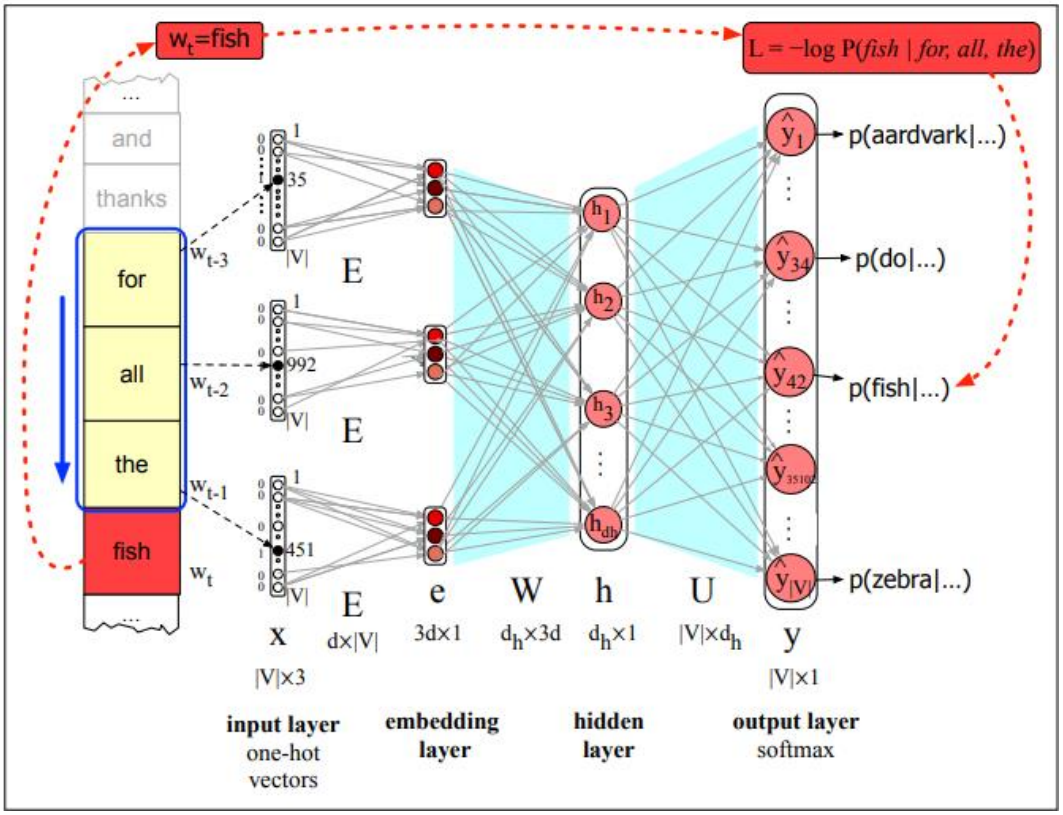
2.3.5.2.4 Training Neural Language Model

Setelah melihat cara melatih jaringan saraf generik, berikut tentang arsitektur untuk melatih model bahasa saraf, menyetel parameter θ = **E**, **W**, **U**, **b**.

Untuk beberapa tugas, boleh saja untuk melakukan *freeze* pada *embedding layer* **E** dengan nilai awal *word2vec*. Pembekuan berarti menggunakan *word2vec* atau algoritma pra-pelatihan lainnya untuk menghitung matriks embedding awal **E**,

dan kemudian mempertahankannya secara konstan sementara yang hanya dimodifikasi **W**, **U**, dan **b**, tidak memperbarui **E** selama pelatihan model bahasa(*language model*). Namun, mempelajari embeddings secara bersamaan dengan melatih jaringan, berguna ketika tugas dari network yang dirancang untuk (seperti klasifikasi sentimen, terjemahan, atau penguraian) menempatkan batasan kuat pada apa yang membuat representasi yang baik untuk kata-kata.

Bagaimana melatih seluruh model termasuk **E**, yaitu mengatur semua parameter θ = **E**, **W**, **U**, **b**, akan dilakukan melalui *gradient descent*, menggunakan *error backpropagation* pada komputasi grafis untuk menghitung *gradient*. Dengan demikian, pelatihan tidak hanya menetapkan bobot **W** dan **U** dari network, tetapi juga memprediksi kata-kata yang akan datang, mempelajari *embeddings* **E** untuk setiap kata yang paling baik memprediksi kata-kata yang akan datang.



Gambar 2.3.17 Melakukan pembelajaran penuh kembali ke embeddings, Matriks embedding **E** dibagi di antara 3 kata konteks.

Gambar 2.3.17 menunjukkan pengaturan untuk ukuran jendela kata konteks N=3. Input **x** terdiri dari 3 *one-hot vector*, terhubung sepenuhnya ke layer *embedding* melalui 3 instantiasi dari *embedding matrix* **E**. Dikondisikan untuk tidak

mempelajari matriks bobot terpisah untuk memetakan masing-masing dari 3 kata sebelumnya ke layer proyeksi. Dikondisikan juga untuk satu embedding dictionary **E** yang dibagikan di antara ketiganya. Itu karena seiring waktu, banyak kata yang berbeda akan muncul sebagai w_{t-2} atau w_{t-1}, dan mengkondisikan kasus untuk mewakili setiap kata dengan satu vektor, di mana pun posisi konteksnya. Matriks bobot embedding **E** memiliki kolom untuk setiap kata, setiap vektor kolom berdimensi *d*, dan karenanya memiliki dimensi $d \times |V|$.

Umumnya pelatihan berlangsung dengan memasukkan teks yang sangat panjang sebagai input, menggabungkan semua kalimat, dimulai dengan bobot acak, dan kemudian secara iteratif bergerak melalui teks yang memprediksi setiap kata w_t. Pada setiap kata w_t, digunakan *cross-entropy loss* (*negative log likelihood*). Bentuk umum untuk ini memiliki rumus:

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i, \quad \text{(where } i \text{ is the correct class)} \tag{2.36}$$

Untuk pemodelan bahasa, kelasnya adalah kata-kata dalam kosa kata, jadi $\hat{y}_i$ di sini berarti probabilitas bahwa model menetapkan kata berikutnya yang benar w_t:

$$L_{\text{CE}} = -\log p(w_t | w_{t-1}, \ldots, w_{t-n+1}) \tag{2.37}$$

Pembaruan parameter untuk *stochastic gradient descent* untuk *loss* ini dari langkah *s* ke *s*+1 adalah:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial \left[-\log p(w_t | w_{t-1}, \ldots, w_{t-n+1}) \right]}{\partial \theta} \tag{2.38}$$

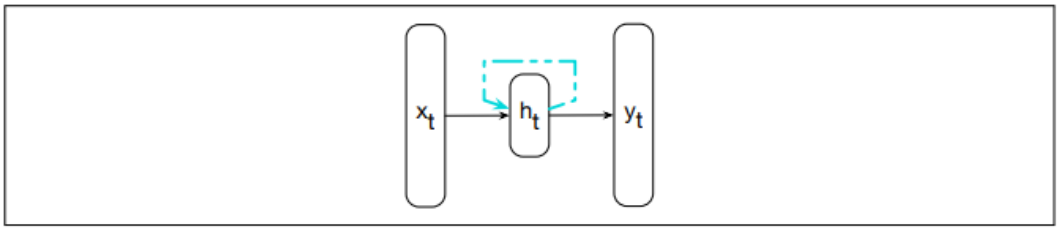
Gradient ini dapat dihitung dalam kerangka jaringan saraf standar manapun yang kemudian akan *backpropagate* melalui θ = **E**, **W**, **U**, **b**.

Melatih parameter untuk meminimalkan loss akan menghasilkan algoritma untuk *language modelling* (prediktor kata) dan juga satu set baru *embeddings* **E** yang dapat digunakan sebagai representasi kata untuk tugas lainnya.

2.3.6 Recurrent Neural Network

Jaringan Saraf Tiruan Berulang atau *Reccurent Neural Network* (RNN) adalah jaringan yang memiliki sebuah putaran di hubungan jaringannya, yang berarti nilai dari suatu unit akan secara langsung atau tak langsung, bergantung pada

output sendiri sebagai inputnya. Walaupun terlihat menjanjikan, jaringan tersebut tetap membutuhkan pengertian dan training yang cukup rumit. Meskipun begitu, dalam kelas umum dari recurrent network ada arsitektur terbatas yang sudah diuji efektivitasnya dalam aplikasinya di bahasa. Adapun kelas tersebut dinamakan sebagai *simple recurrent network* (Elman, 1990). Hidden layer menyertakan koneksi berulang (*reccurent*) sebagai bagian dari inputnya. Artinya, nilai aktivasi hidden layer tergantung pada input saat ini serta nilai aktivasi hidden layer dari langkah waktu sebelumnya.



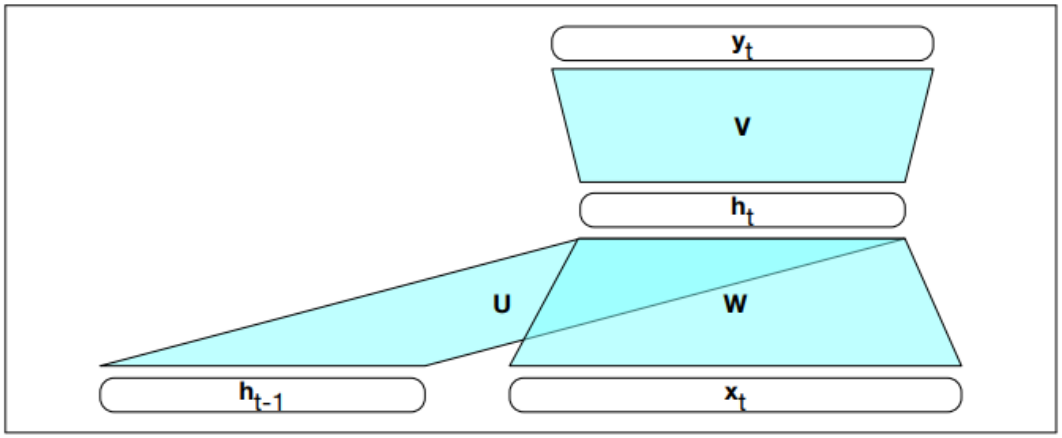
Gambar 2.3.18 Simple reccurent neural network dari (Elman, 1990)

Gambar diatas menjelaskan struktur dasar dari RNN (yang sederhana dan dibatasi). Seperti halnya *feedforward network* biasa, vektor input yang mewakili input saat ini, $\mathbf{x}_t$, dikalikan dengan bobot matriks dan kemudian melewati fungsi aktivasi non-linier untuk menghitung nilai untuk layer hidden unit. Hidden unit ini kemudian digunakan untuk menghitung keluaran yang sesuai, $\mathbf{y}_t$. Berbeda dengan dari pendekatan window-based sebelumnya, sequence diproses dengan menghadirkan satu item pada satu waktu ke jaringan. Subskrip digunakan untuk merepresentasikan waktu, $\mathbf{x}_t$, berarti vektor input **x** pada waktu *t*. Perbedaan utama dari *feedforward network* terletak pada *recurrent link* yang ditunjukkan pada gambar dengan garis putus-putus. *Link* ini menamabah input ke perhitungan pada *hidden layer* dengan nilai *hidden layer* dari titik waktu sebelumnya.

Hidden layer dari *time step* sebelumnya menyediakan bentuk dalam memori, atau konteks, yang mengkodekan pemrosesan sebelumnya dan menginformasikan keputusan yang akan dibuat pada titik waktu selanjutnya. Secara kritis, pendekatan ini tidak memaksakan batasan *fixed-length* pada konteks sebelumnya; konteks yang terkandung dalam *hidden network* sebelumnya dapat mencakup informasi yang meluas kembali ke awal *sequence*.

Menambahkan dimensi temporal ini membuat RNN tampak lebih kompleks daripada arsitektur *non-recurrent*. Namun pada kenyataannya, mereka

tidak jauh berbeda. Mengingat vektor input dan nilai untuk *hidden layer* dari langkah waktu sebelumnya, perhitungan *feedforward* standar masih digunakan. Untuk lebih jelasnya, gambar 2.3.19 yang menjelaskan sifat pengulangan dan bagaimana faktornya ke dalam komputasi pada hidden layer. Perubahan paling signifikan terletak pada rangkaian bobot baru, **U**, yang menghubungkan *hidden layer* dari langkah waktu sebelumnya ke *hidden layer* terkini. Bobot ini menentukan bagaimana jaringan memanfaatkan konteks masa lalu dalam menghitung output untuk input saat ini. Seperti bobot lainnya dalam jaringan, koneksi ini dilatih melalui *backpropagation*.



Gambar 2.3.19 *Simple reccurent neural network* diilustrasikan sebagai sebuah *feedforward network*.

2.3.6.1 Inference di Recurrent Neural Network

Forward Inference (memetakan *sequence* input ke *sequence* output) dalam RNN hampir sama dengan apa yang telah kita lihat dengan *feedforward network*. Untuk menghitung output $\mathbf{y}_t$ terhadap input $\mathbf{x}_t$, kita memerlukan nilai aktivasi untuk hidden layer $\mathbf{h}_t$. Untuk menghitung ini, dikalikan input $\mathbf{x}_t$ dengan matriks bobot **W**, dan *hidden layer* dari langkah waktu sebelumnya $\mathbf{h}_{t-1}$ dengan matriks bobot **U**. Nilai-nilai ini akan ditambahkan bersama-sama dan diteruskan melalui fungsi aktivasi yang sesuai, *g*, sampai pada nilai aktivasi untuk *hidden layer* terkini, $\mathbf{h}_t$. Setelah nilai untuk hidden layer didapat, perhitungan biasa dilakukan untuk menghasilkan vektor keluaran.

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \tag{2.39}$$

$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t) \tag{2.40}$$

Penting di sini untuk berhati-hati dalam menentukan dimensi input, hidden dan output layer, serta matriks bobot untuk memastikan perhitungan ini benar. Dalam input, dimensi hidden dan output layer masing-masing dinyatakan sebagai d_{in}, d_h, dan d_{out}. Mengingat ini, tiga matriks parameter-nya adalah: $W \in \mathbb{R}^{d_h x d_{in}}$, $U \in \mathbb{R}^{d_h x d_h}$, dan $V \in \mathbb{R}^{d_{out} x d_h}$.

Dalam kasus *soft classification* yang umum ditemui, komputasi $\mathbf{y}_t$ terdiri dari komputasi *softmax* yang menyediakan distribusi probabilitas atas kemungkinan positif atas kelas-kelas keluaran.

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \tag{2.41}$$

Fakta bahwa komputasi pada waktu *t* membutuhkan nilai hidden layer dari waktu *t*-1 mengamanatkan algoritma inferensi inkremental yang berjalan dari awal urutan hingga akhir seperti yang diilustrasikan pada Gambar 2.3.20. Sifat sekuensial jaringan berulang sederhana juga dapat dilihat dengan *unrolling* jaringan dalam waktu seperti yang ditunjukkan pada Gambar 9.5. Dalam gambar ini, berbagai lapisan unit disalin untuk setiap langkah waktu untuk menggambarkan bahwa mereka akan memiliki nilai yang berbeda dari waktu ke waktu. Namun, berbagai matriks bobot dibagi sepanjang waktu.

function FORWARDRNN(**x**, *network*) **returns** output sequence **y**

$\mathbf{h}^0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH(**x**) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

Gambar 2.3.20 Forward inference dalam sebuah simple reccurent network. Matriks **U**, **V** dan **W** dibagi sepanjang waktu, sedangkan nilai baru untuk **h** dan **y** dihitung dengan setiap langkah waktu.

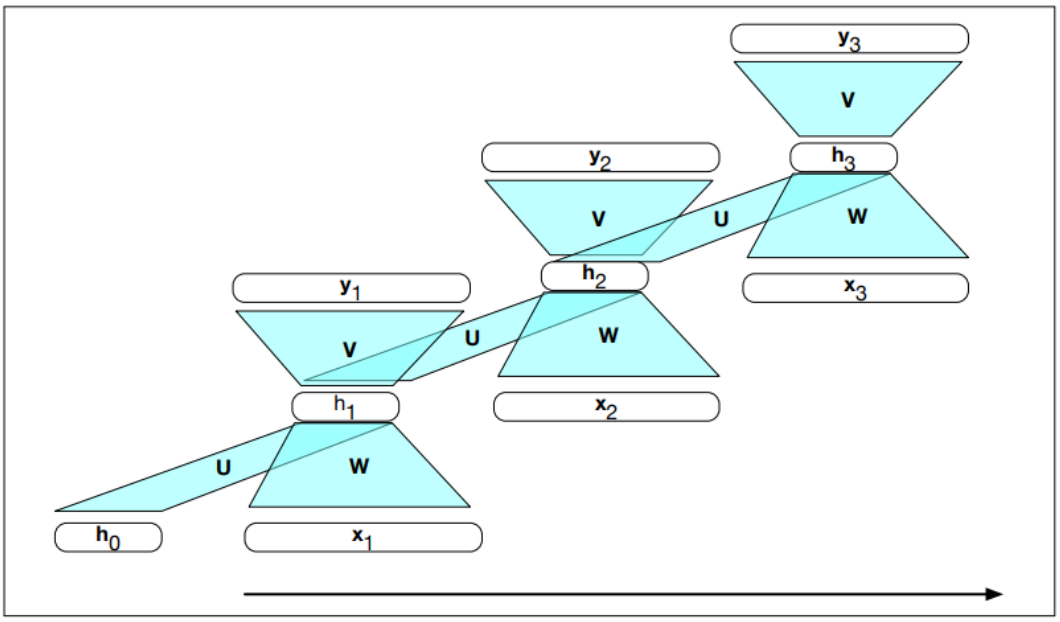
2.3.6.2 Training di Recurrent Neural Network

Seperti halnya jaringan *feedforward*, akan menggunakan set pelatihan, fungsi loss, dan backpropagation untuk mendapatkan gradien yang diperlukan untuk menyesuaikan bobot dalam jaringan berulang ini. Seperti yang ditunjukkan pada Gambar 2.3.19, sekarang kita memiliki 3 set bobot untuk diperbarui: **W**, bobot

dari layer input ke hidden layer, **U**, bobot dari hidden layer sebelumnya ke hidden layer saat ini, dan akhirnya **V**, bobot dari hidden layer ke layer output.

Gambar 2.3.21 menyoroti dua pertimbangan yang tidak perlu kita khawatirkan dengan *backpropagation* di jaringan *feedforward*. Pertama, untuk menghitung fungsi *loss* keluaran pada waktu *t* kita memerlukan *hidden layer* dari waktu *t*-1. Kedua, *hidden layer* pada waktu *t* mempengaruhi baik keluaran pada waktu *t* maupun *hidden layer* pada waktu *t*+1 (dan maka output dan loss pada *t*+1). Oleh karena itu, untuk menilai kesalahan yang timbul pada $\mathbf{h}_t$, kita perlu mengetahui pengaruhnya terhadap keluaran saat ini maupun yang mengikuti.

Menyesuaikan algoritma *backpropagation* untuk situasi ini mengarah ke algoritma *two-pass* untuk melatih bobot dalam RNN. Pada lintasan pertama, kita melakukan forward inference, menghitung $\mathbf{h}_t$, $\mathbf{y}_t$, mengumpulkan loss pada setiap langkah waktu, menyimpan nilai hidden layer pada setiap langkah untuk digunakan pada langkah waktu berikutnya. Pada fase kedua, akan memproses urutan secara terbalik, menghitung *gradient* yang diperlukan saat melanjutkan, menghitung dan menyimpan istilah *error term* untuk digunakan di *hidden layer* untuk setiap langkah mundur dalam waktu. Pendekatan umum ini biasa disebut sebagai *Backpropagation Through Time* (Rumelhart et al., 1986).



Gambar 2.3.21 Recurrent neural network sederhana yang ditampilkan *unrolled* dalam waktu. layer jaringan dihitung ulang untuk setiap langkah waktu, sedangkan bobot **U**, **V**, dan **W** dibagi secara umum di semua langkah waktu.

Untungnya, dengan kerangka kerja komputasi modern dan sumber daya komputasi yang memadai, tidak diperlukan pendekatan khusus untuk melatih RNN. Seperti diilustrasikan pada Gambar 2.3.21, secara eksplisit melakukan *unrolling* recurrent neural network ke dalam komputasi grafis *feedforward* menghilangkan pengulangan eksplisit, memungkinkan bobot jaringan untuk dilatih secara langsung. Dalam pendekatan seperti itu, disediakan template yang menentukan struktur dasar jaringan, termasuk semua parameter yang diperlukan untuk input, output, dan *hidden layer*, matriks bobot, serta fungsi aktivasi dan output yang akan digunakan. Kemudian, ketika disajikan dengan urutan input tertentu, akan dapat menghasilkan *feedforward network* yang berada di kondisi *unrolled* khusus untuk input itu, dan menggunakan grafik itu untuk melakukan *forward inference* atau pelatihan melalui *backpropagation* biasa.

Untuk aplikasi yang melibatkan urutan input yang lebih panjang, seperti pengenalan suara, pemrosesan level karakter, atau streaming input berkelanjutan, *unrolling* untuk seluruh urutan input mungkin tidak dapat dilakukan. Dalam kasus ini, kita dapat *unroll* input ke dalam segmen dengan panjang tetap yang dapat dikelola dan memperlakukan setiap segmen sebagai item pelatihan yang berbeda.

2.3.6.3 Recurrent Neural Network sebagai Language Model

RNN Language Model (Mikolov et al., 2010) memproses *sequence* input satu kata pada satu waktu, mencoba memprediksi kata berikutnya dari kata saat ini dan hidden state sebelumnya. RNN tidak memiliki masalah konteks terbatas yang dimiliki model *n-gram*, karena *hidden state* pada prinsipnya dapat mewakili informasi tentang semua kata sebelumnya sampai ke awal *sequence*.

Forward inference dalam model bahasa *recurrent* berlangsung persis seperti yang dijelaskan dalam Bagian 2.3.6.1. Urutan input $X = [x_1 ;...; x_t ;...; x_N]$ terdiri dari serangkaian *word embeddings* yang masing-masing direpresentasikan sebagai *one-hot vector* ukuran $|V|\times 1$, dan output prediksi, **y**, adalah vektor yang mewakili distribusi probabilitas atas *vocabulary*. Pada setiap langkah, model menggunakan matriks *embedding* kata **E** untuk mengambil *embedding* kata saat ini, dan kemudian menggabungkannya dengan *hidden layer* dari langkah sebelumnya untuk menghitung *hidden layer* baru. *Hidden layer* ini kemudian digunakan untuk

menghasilkan layer output yang dilewatkan melalui layer *softmax* untuk menghasilkan distribusi probabilitas di seluruh *vocabulary*. Artinya, pada waktu *t*:

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{2.42}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{2.43}$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \tag{2.44}$$

Vektor yang dihasilkan dari **Vh** dapat dianggap sebagai kumpulan skor di atas *vocabulary* yang diberikan bukti disediakan dalam **h**. Membagi skor ini melalui *softmax* menormalkan skor menjadi distribusi probabilitas. Probabilitas bahwa kata tertentu *i* dalam vocabulary adalah kata berikutnya diwakili oleh $\mathbf{y}_t[i]$, komponen ke-*i* dari $\mathbf{y}_t$:

$$P(w_{t+1} = i | w_1, \ldots, w_t) = \mathbf{y}_t[i] \tag{2.45}$$

Probabilitas seluruh sequence hanyalah produk dari probabilitas setiap item dalam sequence, di mana kita akan menggunakan $\mathbf{y}_i[w_i]$ untuk mengartikan probabilitas kata yang benar w_i pada langkah waktu *i*.

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1}) \tag{2.46}$$

$$= \prod_{i=1}^{n} \mathbf{y}_i[w_i] \tag{2.47}$$

Untuk melatih RNN sebagai model bahasa, digunakan korpus teks sebagai materi pelatihan, dengan model memprediksi kata berikutnya pada setiap langkah waktu *t*. Model dilatih untuk meminimalkan error dalam memprediksi kata berikutnya yang benar dalam urutan pelatihan, menggunakan fungsi cross-entropy loss sebagai fungsi loss-nya. Cross-entropy loss mengukur perbedaan antara distribusi probabilitas yang diprediksi dan distribusi yang benar.

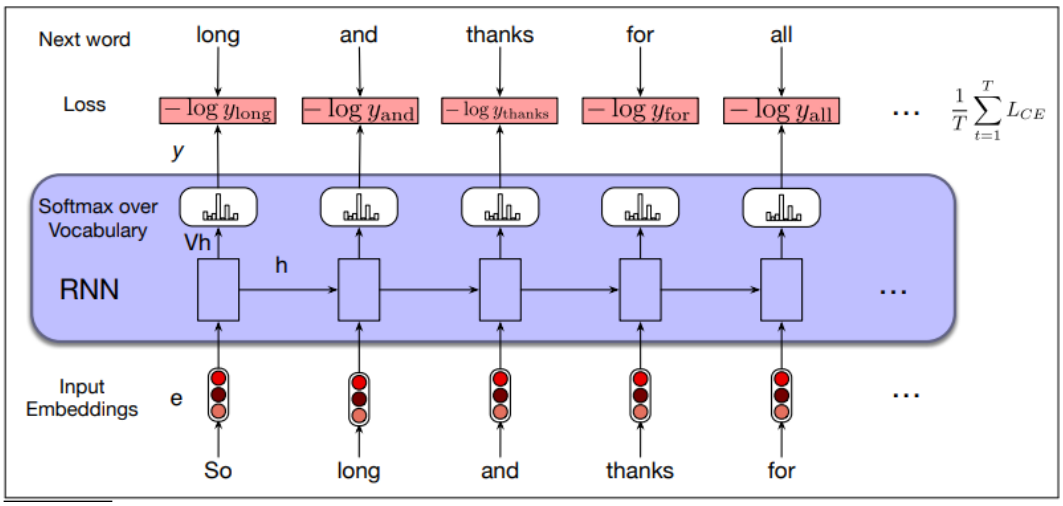
$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \tag{2.48}$$

Dalam kasus pemodelan bahasa, distribusi $\mathbf{y}_t$ yang benar berasal dari mengetahui kata berikutnya. Ini direpresentasikan sebagai *one-hot vector* yang sesuai dengan *vocabulary* di mana entri untuk kata berikutnya yang benar adalah 1, dan semua entri lainnya adalah 0. Dengan demikian, cross-entropy loss untuk pemodelan bahasa ditentukan oleh probabilitas yang diberikan model ke kata

berikutnya yang benar. Jadi pada waktu *t*, CE loss adalah probabilitas log negatif yang diberikan model ke kata berikutnya dalam sequence pelatihan.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{2.49}$$

Jadi pada setiap posisi kata *t* dari input, model mengambil sebagai input urutan token yang benar $w_{1:t}$, dan menggunakannya untuk menghitung distribusi probabilitas atas kemungkinan kata berikutnya sehingga dapat menghitung loss model untuk token berikutnya w_{t+1} . Kemudian pindah ke kata berikutnya, apa yang diprediksi model untuk kata berikutnya akan diabaikan dan sebagai gantinya menggunakan urutan token yang benar $w_{1:t+1}$ untuk memperkirakan probabilitas token w_{t+2}. Idenya bahwa akan selalu diberikan model dengan *history sequence* yang benar untuk memprediksi kata berikutnya (daripada memberi model kasus terbaiknya dari langkah waktu sebelumnya) biasa disebut *teacher forcing*.



Gambar 2.3.22 Training RNN sebagai Language Model

Bobot dalam jaringan disesuaikan untuk meminimalkan CE loss rata-rata selama urutan pelatihan melalui gradient descent. Gambar 2.3.22 mengilustrasikan bentuk pelatihan ini.

Dapat diperhatikan bahwa matriks embedding masukan **E** dan matriks layer akhir **V**, yang mengumpan ke *softmax* keluaran, sangat mirip. Kolom **E** mewakili word embeddings untuk setiap kata dalam vocabulary yang dipelajari selama proses pelatihan dengan tujuan agar kata yang memiliki makna dan fungsi serupa akan memiliki embedding yang serupa juga. Kemudian, karena panjang

embedding ini sesuai dengan ukuran *hidden layer* d_h, bentuk matriks embedding **E** adalah $d_h \times |V|$.

Matriks layer terakhir **V** menyediakan cara untuk menilai kemungkinan setiap kata dalam vocabulary yang diberikan bukti yang ada di *hidden layer* akhir jaringan melalui perhitungan **Vh**. Ini menghasilkan dimensionalitas $|V| \times d_h$. Artinya, baris V menyediakan set kedua dari embeddings kata yang dipelajari yang menangkap aspek relevan dari makna dan fungsi kata. Ini mengarah pada pertanyaan yang jelas – apakah perlu memiliki keduanya? *Weight tying* adalah metode yang menghilangkan redundansi ini dan hanya menggunakan satu set embeddings pada layer input dan softmax. Artinya, kita membuang *V* dan menggunakan *E* baik di awal maupun di akhir perhitungan.

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{2.50}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{2.51}$$

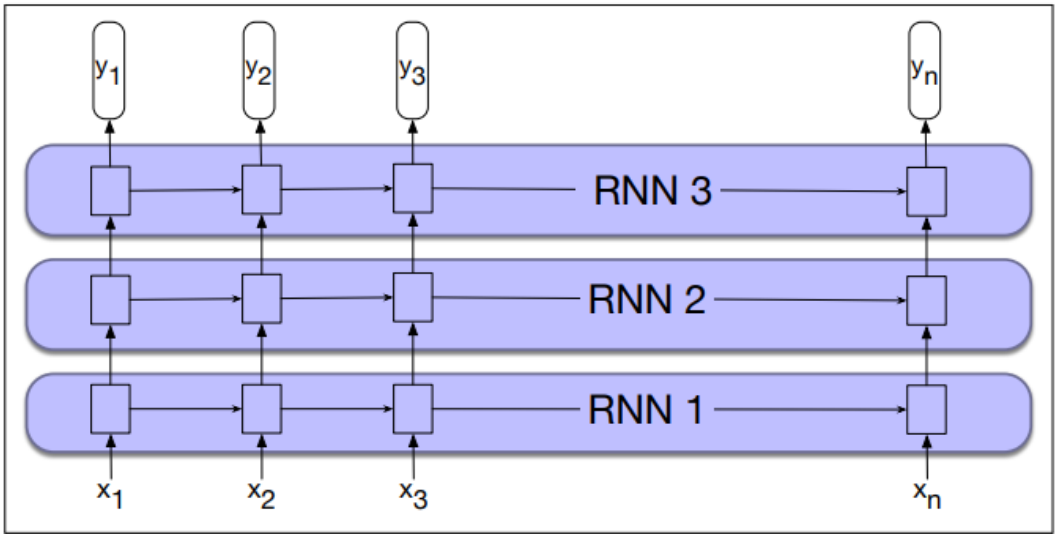
$$\mathbf{y}_t = \text{softmax}(\mathbf{E}^{intercal}\mathbf{h}_t) \tag{2.52}$$

Selain memberikan *perplexity* model yang lebih baik, pendekatan ini secara signifikan mengurangi jumlah parameter yang diperlukan untuk model.

2.3.6.4 Arsitektur pada Recurrent Neural Network

2.3.6.4.1 Stacked Recurrent Neural Network

Dalam beberapa contoh sejauh ini, input ke RNN terdiri dari urutan embedding kata atau karakter (vektor) dan outputnya adalah vektor yang berguna untuk memprediksi kata, tag, atau label sequence. Namun, tidak ada halangan untuk menggunakan seluruh sequence keluaran dari satu RNN sebagai sequence masukan ke sequence lainnya. *Stacked RNNs* terdiri dari beberapa network di mana output dari satu layer berfungsi sebagai input ke layer berikutnya, seperti yang ditunjukkan pada Gambar 2.3.23. Output dari level yang lebih rendah berfungsi sebagai input ke level yang lebih tinggi dengan output dari jaringan terakhir berfungsi sebagai output akhir.



Gambar 2.3.23 Stacked Recurrent Networks.

RNN yang ditumpuk umumnya mengungguli single-layer networks. Salah satu alasan keberhasilan ini tampaknya adalah bahwa jaringan menginduksi representasi pada tingkat abstraksi yang berbeda di seluruh lapisan. Sama seperti tahap awal sistem visual manusia mendeteksi tepian yang kemudian digunakan untuk menemukan wilayah dan bentuk yang lebih besar, layer awal jaringan bertumpuk dapat menginduksi representasi yang berfungsi sebagai abstraksi yang berguna untuk layer selanjutnya—representasi yang mungkin terbukti sulit untuk diinduksi dalam RNN tunggal. Jumlah optimal RNN bertumpuk adalah khusus untuk setiap aplikasi dan untuk setiap set pelatihan. Namun, dengan jumlah tumpukan yang meningkat, biaya pelatihan meningkat juga dengan cepat.

2.3.6.4.2 Bidirectional Recurrent Neural Network

RNN menggunakan informasi dari konteks kiri (sebelumnya) untuk membuat prediksinya pada waktu *t*. Namun dalam banyak pengaplikasiannya memiliki akses ke seluruh sequence input; dalam kasus tersebut menggunakan kata-kata dari konteks di sebelah kanan *t* dapat dilakukan. Salah satu cara untuk melakukannya adalah dengan menjalankan dua RNN terpisah, satu kiri-ke-kanan, dan satu kanan-ke-kiri, dan menggabungkan representasi mereka.

Dalam RNN kiri-ke-kanan yang dibahas sejauh ini, hidden state pada waktu tertentu *t* mewakili semua yang diketahui jaringan tentang sequence hingga

poin itu. State adalah fungsi dari input x_1,..., x_t dan mewakili konteks jaringan di sebelah kiri waktu saat ini.

$$\mathbf{h}_t^f = RNN_{forward}(\mathbf{x}_1, \ldots, \mathbf{x}_t) \tag{2.53}$$

Notasi baru ini $\boldsymbol{h}_t^f$ hanya menyesuaikan dengan hidden state normal pada waktu *t*, mewakili semua jaringan yang telah diperoleh dari sequence sejauh ini.

Untuk memanfaatkan konteks di sebelah kanan input saat ini, kita dapat melatih RNN pada sequence input terbalik. Dengan pendekatan ini, hidden state pada waktu *t* mewakili informasi tentang urutan di sebelah kanan input saat ini:

$$\mathbf{h}_t^b = RNN_{backward}(\mathbf{x}_t, \ldots \mathbf{x}_n) \tag{2.54}$$

Di sini, hidden state $\boldsymbol{h}_t^b$ mewakili semua informasi yang telah kita ketahui tentang sequence dari *t* sampai akhir sequence.

Bidirectional RNN (Schuster & Paliwal, 1997) menggabungkan dua RNN independen, satu di mana input diproses dari awal hingga akhir, dan yang lainnya dari akhir hingga awal. Kemudian menggabungkan dua representasi yang dihitung oleh jaringan menjadi satu vektor yang menangkap konteks kiri dan kanan input pada setiap titik waktu. Di sini akan digunakan titik koma ";" atau simbol $\oplus$ yang mengartikan rangkaian vektor:

$$\begin{aligned} \mathbf{h}_t &= [\mathbf{h}_t^f ; \mathbf{h}_t^b] \\ &= \mathbf{h}_t^f \oplus \mathbf{h}_t^b \end{aligned} \tag{2.55}$$

Bidirectional network akan menggabungkan output dari *forward* dan *backward pass*. Cara sederhana lainnya untuk menggabungkan konteks *forward* dan *backward* termasuk penambahan atau perkalian *element-wise*. Output pada setiap langkah dalam waktu akan menangkap informasi ke kiri dan ke kanan input saat ini. Dalam aplikasi *sequence labeling*, output gabungan ini dapat berfungsi sebagai dasar untuk keputusan pelabelan lokal.

RNN dua arah juga telah terbukti cukup efektif untuk sequence classification. Untuk klasifikasi sequence akan digunakan hidden state akhir dari RNN sebagai input ke pengklasifikasi feedforward berikutnya. Kesulitan dengan pendekatan ini adalah bahwa keadaan akhir secara alami mencerminkan lebih banyak informasi tentang akhir kalimat daripada awalnya. RNN dua arah memberikan solusi sederhana untuk masalah ini; cukup menggabungkan hidden

state akhir dari lintasan maju dan mundur (misalnya dengan penggabungan) dan menggunakannya sebagai input untuk pemrosesan lanjutan.

2.3.6.5 Long Short-Term Memory Network

Dalam praktiknya, cukup sulit untuk melatih RNN untuk tugas-tugas yang memerlukan jaringan untuk memanfaatkan informasi yang jauh dari titik pemrosesan saat ini. Meskipun memiliki akses ke seluruh sequence sebelumnya, informasi yang dikodekan dalam hidden state cenderung cukup lokal, lebih relevan dengan bagian terbaru dari urutan input dan keputusan terbaru. Namun informasi yang jauh sangat penting untuk banyak aplikasi bahasa. Perhatikan contoh berikut dalam konteks pemodelan bahasa.

(2.56) The flights the airline was cancelling were full

Menetapkan probabilitas tinggi untuk (*was*) mengikuti (*airline*) sangatlah mudah karena (*airline*) memberikan konteks lokal yang kuat untuk penerimaan tunggal. Namun, menetapkan probabilitas yang tepat untuk (*were*) cukup sulit, bukan hanya karena jamak (*flights*) cukup jauh, tetapi juga karena konteks intervensi melibatkan konstituen tunggal. Idealnya, sebuah jaringan harus dapat menyimpan informasi yang jauh tentang jamak (*flights*) sampai dibutuhkan, sambil tetap memproses bagian-bagian perantara dari urutan dengan benar.

Salah satu alasan ketidakmampuan RNN untuk meneruskan informasi penting adalah bahwa *hidden layer*, dan, dengan perluasan, bobot yang menentukan nilai di *hidden layer*, diminta untuk melakukan dua tugas secara bersamaan: memberikan informasi yang berguna untuk keputusan saat ini, dan memperbarui dan meneruskan informasi yang diperlukan untuk keputusan di masa mendatang.

Kesulitan kedua dengan pelatihan RNN muncul dari kebutuhan untuk melakukan *backpropagate* sinyal *error* melalui waktu. Ingat dari Bagian 2.3.6.2 bahwa *hidden layer* pada waktu *t* berkontribusi pada *loss* pada langkah waktu berikutnya karena mengambil bagian dalam perhitungan itu. Akibatnya, selama pelatihan *backward pass*, *hidden layer* menjadi subjek perkalian berulang, seperti yang ditentukan oleh panjang urutan. Hasil yang sering dari proses ini adalah bahwa gradient akhirnya didorong ke nol, suatu situasi yang disebut sebagai *vanishing gradient*.

Untuk mengatasi masalah ini, arsitektur jaringan yang lebih kompleks telah dirancang untuk secara eksplisit mengelola tugas mempertahankan konteks yang relevan dari waktu ke waktu, dengan memungkinkan jaringan untuk belajar melupakan informasi yang tidak lagi diperlukan dan mengingat informasi yang diperlukan untuk keputusan yang akan datang.

Ekstensi semacam itu yang paling umum digunakan untuk RNN adalah jaringan *Long short-term memory* (LSTM) (Hochreiter & Schmidhuber, 1997). LSTM membagi masalah manajemen konteks menjadi dua sub-masalah: menghapus informasi yang tidak lagi diperlukan dari konteks, dan menambahkan informasi yang mungkin diperlukan untuk pengambilan keputusan nanti. Kunci untuk memecahkan kedua masalah tersebut adalah mempelajari bagaimana mengelola konteks ini daripada mengkodekan strategi secara keras ke dalam arsitektur. LSTM mencapai ini dengan terlebih dahulu menambahkan layer konteks eksplisit ke arsitektur (selain hidden layer berulang yang biasa), dan melalui penggunaan unit saraf khusus yang memanfaatkan gerbang atau *gates* untuk mengontrol aliran informasi masuk dan keluar dari unit yang terdiri dari network layer. Gerbang ini diimplementasikan melalui penggunaan bobot tambahan yang beroperasi secara berurutan pada input, dan *hidden layer* sebelumnya, dan layer konteks sebelumnya.

Gerbang dalam LSTM memiliki pola desain yang sama; masing-masing terdiri dari *feedforward layer*, diikuti oleh fungsi aktivasi sigmoid, diikuti oleh perkalian *pointwise* dengan layer yang dipagari. Pilihan sigmoid sebagai fungsi aktivasi muncul dari kecenderungannya untuk mendorong outputnya ke 0 atau 1. Menggabungkannya dengan perkalian *pointwise* memiliki efek yang mirip dengan *binary mask*. Nilai-nilai di layer yang dipagari yang sejajar dengan nilai-nilai di dekat 1 di *mask* dilewatkan hampir tidak berubah; nilai yang sesuai dengan nilai yang lebih rendah pada dasarnya dihapus.

Gerbang pertama yang akan kita pertimbangkan adalah *forget gate*. Tujuan dari gerbang ini untuk menghapus informasi dari konteks yang sudah tidak diperlukan lagi. *Forget gate* menghitung jumlah pembobotan dari *hidden layer state* sebelumnya dan input saat ini dan melewatkannya melalui sigmoid. *Mask* ini kemudian dikalikan *element-wise* dengan vektor konteks untuk menghapus

informasi dari konteks yang tidak lagi diperlukan. Perkalian elemen dua vektor (diwakili oleh operator $\odot$, dan kadang-kadang disebut produk *Hadamard*) adalah vektor dengan dimensi yang sama dengan dua vektor input, di mana setiap elemen *i* adalah produk dari elemen *i* dalam dua vektor input:

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t) \tag{2.57}$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t \tag{2.58}$$

Tugas berikutnya adalah menghitung informasi aktual yang perlu diekstrak dari hidden state sebelumnya dan input saat ini—komputasi dasar yang sama yang telah kita gunakan untuk semua jaringan berulang sebelumnya.

$$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t) \tag{2.59}$$

Selanjutnya, membuat *mask* untuk *add gate* untuk memilih informasi yang akan ditambahkan ke konteks saat ini.

$$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t) \tag{2.60}$$

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t \tag{2.61}$$

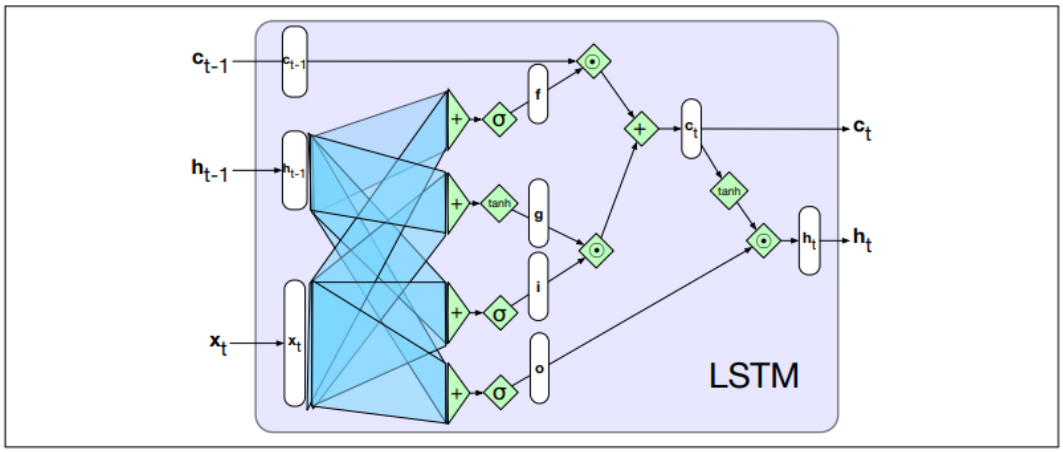
Selanjutnya, menambahkan ini ke vektor konteks yang dimodifikasi untuk mendapatkan vektor konteks yang baru.

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t \tag{2.62}$$

Gerbang terakhir yang akan kita gunakan adalah *output gate* yang digunakan untuk memutuskan informasi apa yang diperlukan untuk hidden state saat ini (sebagai lawan dari informasi apa yang perlu dipertahankan untuk keputusan di masa mendatang).

$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t) \tag{2.63}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{2.64}$$



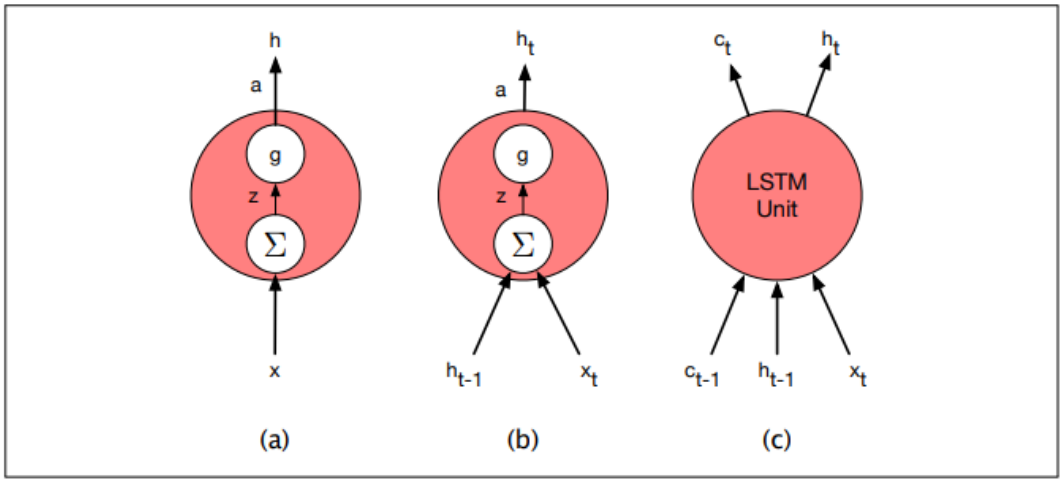
Gambar 2.3.24 Satu unit LSTM ditampilkan sebagai komputasi grafis. Input ke setiap unit terdiri dari input saat ini, *x*, hidden state sebelumnya, h_{t-1}, dan konteks sebelumnya, c_{t-1}. Outputnya adalah hidden state baru, h_t dan konteks yang diperbarui, c_t.

Gambar 2.3.24 mengilustrasikan perhitungan lengkap untuk satu unit LSTM. Mengingat bobot yang sesuai untuk berbagai gerbang, LSTM menerima sebagai input layer konteks, dan hidden layer dari langkah waktu sebelumnya, bersama dengan vektor input saat ini. Kemudian menghasilkan konteks yang diperbarui dan hidden vektor sebagai output. Hidden layer, h_t, dapat digunakan sebagai input ke layer berikutnya dalam RNN yang ditumpuk, atau untuk menghasilkan output untuk lapisan akhir network.

2.3.6.5.1 Gated Unit, Layer dan Network

Unit saraf yang digunakan dalam LSTM jelas jauh lebih kompleks daripada yang digunakan dalam jaringan *feedforward* dasar. Untungnya, kompleksitas ini dikemas dalam unit pemrosesan dasar, memungkinkan untuk mempertahankan modularitas dan dengan mudah bereksperimen dengan arsitektur yang berbeda. Untuk melihat ini, perhatikan Gambar 2.3.25 yang menggambarkan input dan output yang terkait dengan setiap jenis unit.

Di paling kiri, (a) adalah unit feedforward biasa di mana satu set bobot dan satu fungsi aktivasi menentukan outputnya, dan ketika diatur dalam layer tidak ada koneksi di antara unit-unit di layer. Selanjutnya, (b) mewakili unit dalam jaringan berulang sederhana. Sekarang ada dua input dan satu set bobot tambahan yang menyertainya. Namun, masih ada satu fungsi aktivasi dan output.



Gambar 2.3.25 *Basic neural units* yang digunakan dalam *feedforward*, *simple recurrent networks* (SRN), dan *long short-term memory* (LSTM).

Modularitas ini adalah kunci kekuatan dan penerapan luas unit LSTM. Unit LSTM (atau variasi lain, seperti GRU) dapat diganti ke salah satu arsitektur jaringan yang dijelaskan dalam Bagian 2.3.6.4. Seperti halnya RNN sederhana, jaringan berlapis-lapis yang menggunakan *gated units* dapat dilakukan *unrolled* ke dalam *deep feedforward network* dan dilatih dengan cara biasa dengan *backpropagation*.

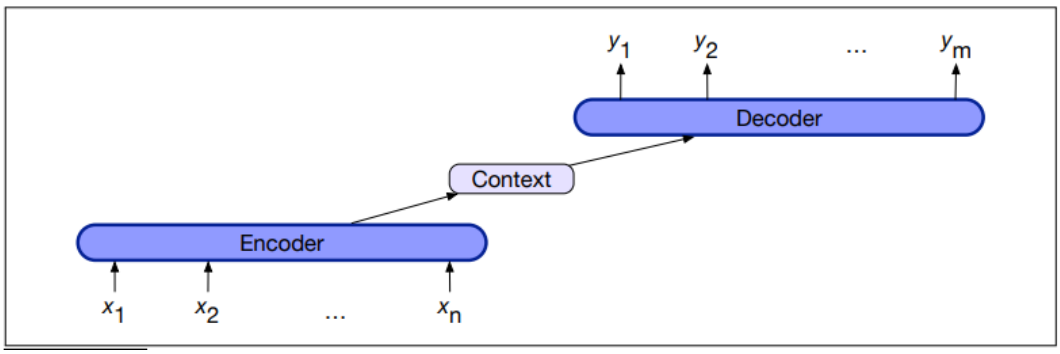
2.3.7 Machine Translation dengan Encoder-Decoder

Model dasar *encoder-decoder* atau *sequence to sequence* sudah menjadi dasar untuk algoritma pada penerjemahan mesin modern saat ini. Umum digunakan untuk berbagai jenis *sequence modelling* yang mana output *sequence* merupakan fungsi kompleks dari keseluruhan input *sequencer*. Kata-kata yang akan menjadi target tidak perlu diharuskan dengan apa yang seharusnya ada di bahasa sumber dalam angkanya atau urutannya. Hal tersebut yang menjadikan tugas *machine translation* berbeda dari yang lain, yang mana dalam kasusnya *mapping* akan dilakukan di sebuah *sequence* kata dari input atau token ke *sequence of tags*, dengan tidak serta merta melakukannya langsung dari kata-kata secara individual. Hal tersebut juga menjadikan model jaringan *encoder-decoder* sangat lihai untuk permasalahan yang membutuhkan *sequence mapping* yang rumit.

2.3.7.1 Model Encoder-Decoder

Encoder-decoder network, atau *sequence to sequence network*, adalah model yang mampu untuk menghasilkan konteks yang sesuai, panjang fleksibel, dari *sequence* output. Encoder-decoder network sudah banyak diaplikasikan ke berbagai penggunaan termasuk penerjemahan mesin, *summarization*, penjawab pertanyaan, dan dialog.

Ide kunci yang mendasari jaringan ini adalah penggunaan jaringan *encoder* yang mengambil urutan input dan menciptakan representasi kontekstual, sering disebut konteks. Representasi ini kemudian diteruskan ke *decoder* yang menghasilkan sequence output tugas tertentu. Gambar 2.3.26 mengilustrasikan arsitekturnya.



Gambar 2.3.26 Arsitektur *encoder-decoder*. Konteks adalah fungsi dari representasi tersembunyi dari input, dan dapat digunakan oleh dekoder dalam berbagai cara.

Jaringan encoder-decoder terdiri atas 3 komponen:

1. **Encoder**, yang menerima input *sequence*, x_1^n, dan menghasilkan *sequence* yang sesuai dari representasi kontekstual, h_1^n. *Encoder* dapat diaplikasikan dengan berbagai arsitektur *sequence* manapun (LSTM, *convolutional network*).
2. **Context vector**, *c*, yang merupakan fungsi dari h_1^n, dan yang menyampaikan esensi input ke *decoder*.
3. **Decoder**, yang menerima *c* sebagai input dan menghasilkan sebuah *sequence* dengan panjang menyesuaikan dari hidden state h_1^m, yang mana *sequence* yang sesuai dari state output y_1^m, dapat diperoleh. Sama seperti *encoder*, *decoder* dapat diwujudkan dengan berbagai arsitektur *sequence* apapun.

2.3.7.2 Model Encoder-Decoder dengan RNN

Pada dasarnya mengulang kembali apa yang dilakukan untuk *language model* di *neural network*, yaitu menghitung $p(y)$ sebagai kemungkinan muncul dari sequence y.

$$p(y) = p(y_1)p(y_2|y_1)p(y_3|y_1,y_2)...P(y_m|y_1,...,y_{m-1}) \quad (2.65)$$

Pada waktu t tertentu, akan diberikan *prefix* dari token t-1 melalui LM, menggunakan *forward inference* untuk memproduksi sebuah *sequence* dari *hidden state*, yang berakhir pada *hidden state* yang berada di posisi terakhir kata dari *prefix*. *Final hidden state* dari *prefix* akan digunakan sebagai poin mulai untuk menghasilkan token selanjutnya.

Formalnya, jika g adalah *activation function* seperti *tanh* atau ReLU, sebuah fungsi dari input pada waktu t dan *hidden state* pada waktu t-1, dan f sebagai *softmax* terhadap set dari *vocabulary* kemungkinan dari kata, dan pada waktu t output $\mathbf{y}_t$ dan *hidden state* $\mathbf{h}_t$ dihitung sebagai berikut:

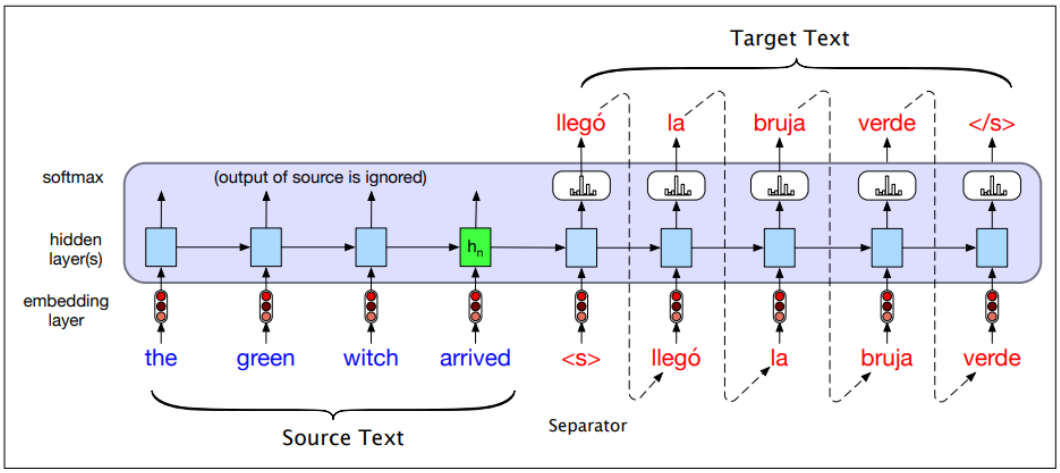
$$\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t) \quad (2.66)$$

$$\mathbf{y}_t = f(\mathbf{h}_t) \quad (2.67)$$

Hanya diperlukan membuat satu perubahan kecil untuk mengubah model bahasa dengan generasi *autoregressive* ini menjadi model terjemahan yang dapat menerjemahkan dari teks sumber dalam satu bahasa ke teks target dengan cepat: tambahkan penanda pemisah kalimat di akhir teks sumber, lalu cukup gabungkan teks target.

Dengan memanggil teks sumber x dan teks target y, perhitungan probabilitas $p(y|x)$ sebagai berikut:

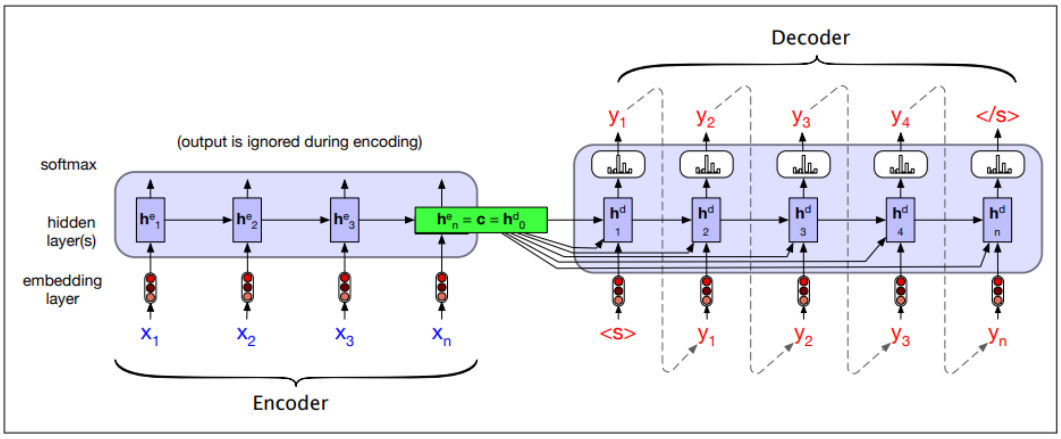
$$p(y|x) = p(y_1|x)p(y_2|y_1,x)p(y_3|y_1,y_2,x)...P(y_m|y_1,...,y_{m-1},x) \quad (2.68)$$



Gambar 2.3.27 Menerjemahkan satu kalimat (waktu inferensi) dalam versi RNN dasar dari pendekatan *encoder-decoder* ke terjemahan mesin. Kalimat sumber dan target digabungkan dengan token pemisah di antaranya, dan *decoder* menggunakan informasi konteks dari *hidden state* terakhir dari *encoder*.

Gambar 2.3.27 menunjukkan susunan dari versi sederhana dari model *encoder-decoder*. Menunjukkan bahwa teks sumber dalam bahasa inggris "the green witch arrived", sebuah token pemisah kalimat <s>, dan teks target dalam bahasa spanyol "*llego la bruja verde*". Untuk menterjemahkan sebuah teks sumber, akan dijalankan melalui jaringan yang melakukan forward inference untuk menghasilkan hidden state hingga mencapai akhir sumber. Kemudian memulai *autoregressive generation*, meminta sebuah kata dalam konteks hidden layer dari akhir input sumber serta penanda akhir kalimat. Kata-kata berikutnya dikondisikan pada hidden state sebelumnya dan penyematan untuk kata terakhir yang dihasilkan.

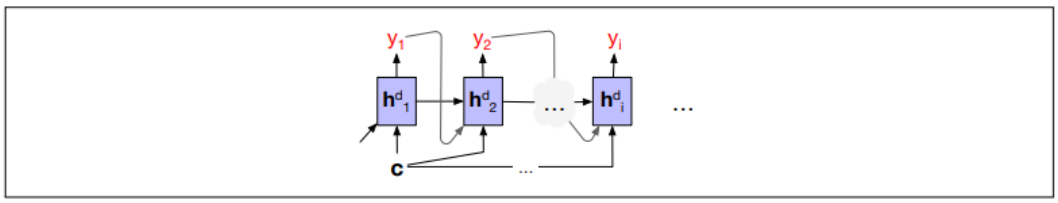
Bentuk yang lebih formal dan ter-generalisasikan dari model ini ada pada Gambar 10.5. (Untuk membantu menjaga semuanya tetap jelas, akan digunakan superskrip *e* dan *d* di mana diperlukan untuk membedakan *hidden state* dari *encoder* dan *decoder*.) Elemen-elemen jaringan di sebelah kiri memproses *sequence* input *x* dan meliputi dari *encoder*. Sementara di ilustrasi yang ditunjukkan yang disederhanakan hanya menunjukkan satu lapisan jaringan untuk *encoder*, arsitektur bertumpuk adalah norma-nya, di mana status keluaran dari layer atas tumpukan diambil sebagai representasi akhir. Desain *encoder* yang banyak digunakan memanfaatkan *biLSTM* bertumpuk di mana *hidden state* dari layer atas dari *forward* dan *backward pass* digabungkan untuk memberikan representasi kontekstual untuk setiap langkah waktu.



Gambar 2.3.28 Bentuk versi mendekati formal untuk menerjemahkan sebuah kalimat pada inference time dalam basis umum RNN untuk arsitektur *encoder-decoder*. *Hidden state* akhir dari *encoder* RNN, h_n^e, berlaku sebagai konteks untuk *decoder* di bagiannya sebagai h_0^d dalam *decoder* RNN.

Tujuan utama dari *encoder* adalah untuk menghasilkan representasi input yang dikontekstualisasikan. Representasi ini diwujudkan dalam *hidden state* terakhir dari *encoder*, h_n^e. Representasi ini, juga disebut *c* untuk konteks, kemudian diteruskan ke dekoder.

Jaringan *decoder* di sebelah kanan mengambil state ini dan menggunakannya untuk menginisialisasi *hidden state* pertama dari *decoder*. Artinya, sel RNN dekoder pertama menggunakan *c* sebagai *hidden state* sebelumnya h_0^d. Dekoder secara otomatis menghasilkan *sequence* output, elemen pada satu waktu, hingga penanda akhir urutan dihasilkan. Setiap *hidden state* dikondisikan pada *hidden state* sebelumnya dan output yang dihasilkan pada state sebelumnya.



Gambar 2.3.29 Mengizinkan setiap *hidden state* dari *decoder* (bukan hanya keadaan *decoder* pertama) dipengaruhi oleh konteks *c* yang dihasilkan oleh *encoder*.

Salah satu kelemahan dari pendekatan ini seperti yang dijelaskan sejauh ini adalah bahwa pengaruh vektor konteks, *c*, akan berkurang ketika sequence keluaran dihasilkan. Solusinya adalah dengan membuat vektor konteks *c* tersedia pada setiap langkah dalam proses *decoding* dengan menambahkannya sebagai

parameter untuk perhitungan *hidden state* saat ini, menggunakan persamaan berikut (diilustrasikan pada Gambar 2.3.29):

$$\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \tag{2.69}$$

Dapat dilihat persamaan lengkap di versi dekoder ini dalam model *encoder-decoder* dasar, dengan konteks yang tersedia di setiap langkah waktu *decoding*. Ingat bahwa *g* adalah stand-in untuk beberapa rasa RNN dan $\hat{y}_{t-1}$ adalah *embedding* untuk output sampel dari *softmax* pada langkah sebelumnya:

$$\begin{aligned} \mathbf{c} &= \mathbf{h}_n^e \\ \mathbf{h}_0^d &= \mathbf{c} \\ \mathbf{h}_t^d &= g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \\ \mathbf{z}_t &= f(\mathbf{h}_t^d) \\ y_t &= \text{softmax}(\mathbf{z}_t) \end{aligned} \tag{2.70}$$

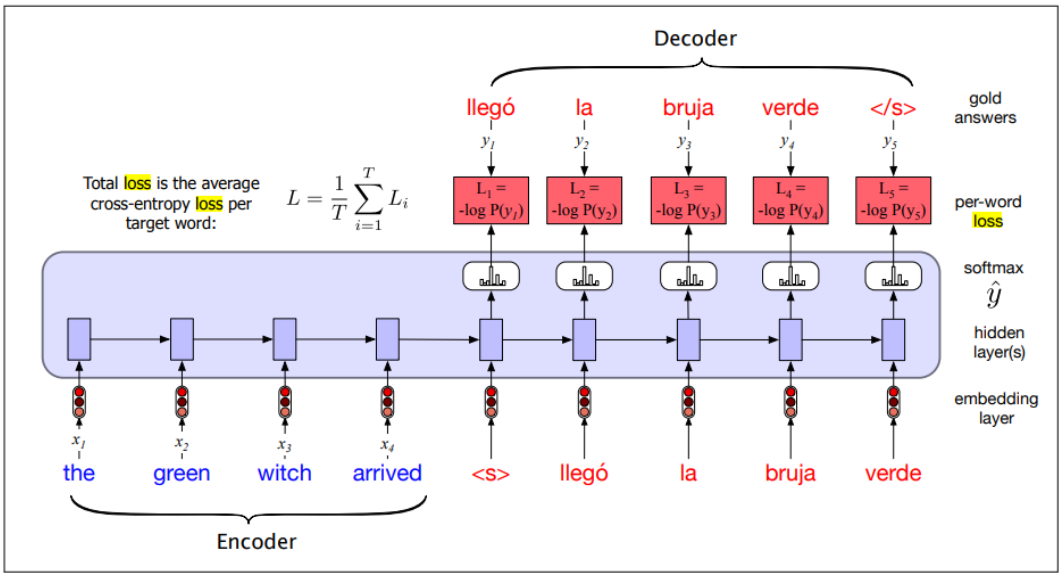
Seperti yang ditunjukkan sebelumnya, keluaran *y* pada setiap langkah waktu terdiri dari perhitungan *softmax* atas himpunan keluaran yang mungkin (*vocabulary*, dalam kasus pemodelan bahasa atau MT). Dilakukan penghitungan output yang paling mungkin pada setiap langkah waktu dengan mengambil *argmax* di atas output *softmax*:

$$\hat{y}_t = \text{argmax}_{w \in V} P(w|x, y_1 \ldots y_{t-1}) \tag{2.71}$$

2.3.7.2.1 Training Model Encoder-Decoder

Arsitektur *encoder-decoder* dilatih dari ujung ke ujung, sama seperti model bahasa RNN. Setiap contoh pelatihan adalah tupel dari string berpasangan, sumber dan target. Digabungkan dengan token pemisah, pasangan sumber-target ini sekarang dapat berfungsi sebagai data pelatihan.

Untuk MT, data pelatihan biasanya terdiri dari kumpulan kalimat dan terjemahannya. Ini dapat diambil dari kumpulan data standar pasangan kalimat yang selaras. Setelah memiliki set pelatihan, pelatihan itu sendiri berlanjut seperti model bahasa berbasis RNN lainnya. Jaringan diberikan teks sumber dan kemudian dimulai dengan token pemisah dilatih secara autoregressive untuk memprediksi kata berikutnya, seperti yang ditunjukkan pada Gambar 10.7.



Gambar 2.3.30 Melatih pendekatan *encoder-decoder* RNN dasar untuk terjemahan mesin. Perhatikan bahwa dalam dekoder disini biasanya tidak menyebarkan output *softmax* model $\hat{y}_t$, tetapi menggunakan *teacher forcing* untuk memaksa setiap input ke nilai terbaik yang benar untuk pelatihan. Penghitungan distribusi output *softmax* lebih dari $\hat{y}$ dalam dekoder untuk menghitung *loss* pada setiap token, yang kemudian dapat dirata-ratakan untuk menghitung *loss* untuk kalimat tersebut.

Perhatikan perbedaan antara pelatihan (Gambar. 2.3.30) dan inferensi (Gambar. 2.3.27) sehubungan dengan output pada setiap langkah waktu. Dekoder selama inferensi menggunakan perkiraan keluarannya sendiri $\hat{y}_t$ sebagai masukan untuk langkah waktu berikutnya x_{t+1}. Dengan demikian *decoder* akan cenderung semakin menyimpang dari kalimat target terbaik karena terus menghasilkan lebih banyak token. Oleh karena itu, dalam pelatihan, lebih umum menggunakan *teacher forcing* di dekoder. *Teacher forcing* berarti akan memaksa sistem untuk menggunakan token target terbaik dari pelatihan sebagai input berikutnya x_{t+1}, daripada membiarkannya mengandalkan output dekoder (kemungkinan error) $\hat{y}_t$. Ini mempercepat pelatihan.

2.3.7.3 Mekanisme Attention

Kesederhanaan model *encoder-decoder* didasari oleh pemisahan bersih *encoder*—yang membangun representasi teks sumber—dari *decoder*, yang menggunakan konteks ini untuk menghasilkan teks target. Dalam model seperti yang telah dijelaskan sejauh ini, vektor konteks ini adalah h_n, *hidden state* dari langkah waktu (*n*-th) terakhir dari teks sumber. *Hidden state* terakhir ini dengan

demikian bertindak sebagai *bottleneck*: yang mana harus benar-benar mewakili segala sesuatu tentang makna teks sumber, karena satu-satunya hal yang diketahui dekoder tentang teks sumber adalah apa yang ada dalam vektor konteks ini (Gbr. 10.8). Informasi di awal kalimat, terutama untuk kalimat yang panjang, mungkin tidak terwakili dengan baik dalam vektor konteks.

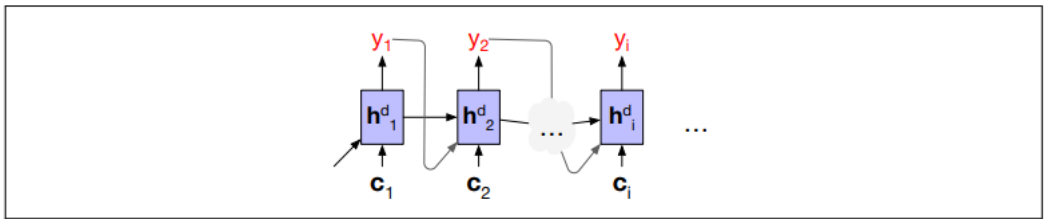
Mekanisme *attention* adalah solusi untuk masalah *bottleneck*, cara yang memungkinkan *decoder* mendapatkan informasi dari semua *hidden state encoder*, bukan hanya *hidden state* terakhir.

Dalam mekanisme *attention*, seperti dalam model dasar *encoder-decoder*, vektor konteks *c* adalah vektor tunggal yang merupakan fungsi dari *hidden state encoder*, yaitu, $c = f(h_1^e \ldots h_n^e)$. Karena jumlah *hidden state* bervariasi dengan ukuran input, tidak dapat menggunakan seluruh *tensor* vektor *hidden state encoder* secara langsung sebagai konteks untuk dekoder.

Ide dari *attention* adalah untuk membuat vektor *c* dengan panjang tetap tunggal dengan mengambil jumlah pembobotan dari semua *hidden state encoder*. Bobotnya fokus pada ('mengikuti') bagian tertentu dari teks sumber yang relevan dengan token yang sedang diproduksi oleh *decoder*. *Attention* dengan demikian menggantikan vektor konteks statis dengan vektor yang secara dinamis diturunkan dari *hidden state encoder*, berbeda untuk setiap token dalam *decoding*.

Vektor konteks ini, $\mathbf{c}_i$, dihasilkan lagi dengan setiap langkah *decoding i* dan memperhitungkan semua *hidden state encoder* dalam derivasinya. Kami kemudian membuat konteks ini tersedia selama *decoding* dengan mengkondisikan perhitungan *hidden state decoder* saat ini di atasnya (bersama dengan *hidden state* sebelumnya dan output sebelumnya yang dihasilkan oleh *decoder*), seperti yang terlihat dalam persamaan ini (dan Gambar 2.3.31):

$$\mathbf{h}_i^d = g(\hat{y}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i) \tag{2.72}$$



Gambar 2.3.31 Mekanisme *attention* memungkinkan setiap *hidden state* dari *decoder* untuk melihat konteks yang berbeda, dinamis, yang merupakan fungsi dari semua *hidden state encoder*.

Langkah pertama dalam menghitung $\mathbf{c}_i$ adalah menghitung seberapa banyak fokus pada setiap status *encoder*, seberapa relevan setiap state *encoder* dengan state *decoder* yang ditangkap dalam h_{i-1}^{d}. Relevansi ditentukan dengan menghitung— pada setiap state *i* selama *decoding*—skor (h_{i-1}^{d}, h_{j}^{e}) untuk setiap state *encoder j*.

Skor yang paling sederhana, yang disebut *dot-product attention*, menerapkan relevansi sebagai pengukur kesamaan: mengukur seberapa mirip *hidden state* dekoder dengan *hidden state* enkoder, dengan menghitung *dot-product* di antara keduanya:

$$score(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{j}^{e}) = \mathbf{h}_{i-1}^{d} \cdot \mathbf{h}_{j}^{e} \tag{2.72}$$

Skor yang dihasilkan dari *dot-product* ini merupakan skalar yang mencerminkan derajat kemiripan antara kedua vektor. Vektor skor ini di semua *hidden state encoder* memberi kita relevansi setiap state *encoder* dengan langkah *decoder* saat ini.

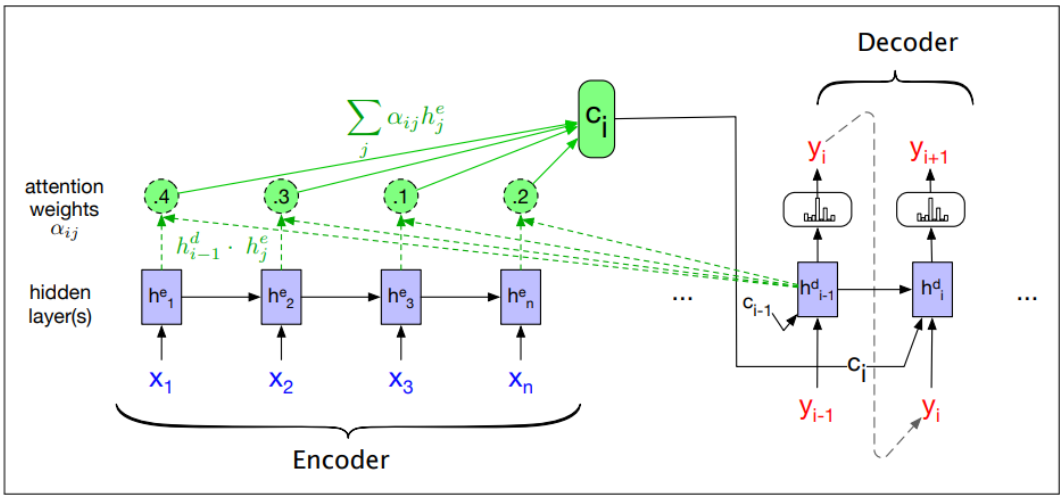
Untuk memanfaatkan skor ini, pertama akan menormalkannya dengan *softmax* untuk membuat vektor bobot, α_{ij}, yang memberi informasi terkait relevansi proporsional dari setiap *hidden state encoder j* ke state *hidden decoder* sebelumnya, h_{i-1}^{d}.

$$\begin{aligned} \alpha_{ij} &= \text{softmax}(score(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{j}^{e}) \ \forall j \in e) \\ &= \frac{\exp(score(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{j}^{e}))}{\sum_{k} \exp(score(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{k}^{e}))} \end{aligned} \tag{2.73}$$

Akhirnya, mengingat distribusi dalam α, dapat dihitung vektor konteks dengan panjang tetap untuk state *decoder* saat ini dengan mengambil rata-rata weight atas semua *hidden state encoder*.

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j^e \tag{2.74}$$

Dengan ini, didapatlah vektor konteks dengan panjang tetap yang memperhitungkan informasi dari seluruh state *encoder* yang diperbarui secara dinamis untuk mencerminkan kebutuhan *decoder* pada setiap langkah *decoding*. Gambar 10.10 mengilustrasikan jaringan encoder-decoder dengan perhatian, dengan fokus pada perhitungan satu vektor konteks ci.



Gambar 2.3.32 Sebuah sketsa jaringan *encoder-decoder* dengan *attention*, dengan fokus pada perhitungan $\mathbf{c}_i$. Nilai konteks $\mathbf{c}_i$ merupakan salah satu input untuk komputasi h_i^d. Ini dihitung dengan mengambil jumlah bobot dari semua *hidden state encoder*, masing-masing dibobotkan dengan *dot product* dengan *hidden state decoder* sebelumnya h_{i-1}^d.

Akan memungkinkan untuk membuat fungsi penilaian yang lebih canggih untuk model attention. Alih-alih *attention dot product* sederhana, bisa menggunakan fungsi yang lebih kuat yang menghitung relevansi setiap *hidden state encoder* ke *hidden state decoder* dengan membuat parameter skor dengan kumpulan bobotnya sendiri, $\mathbf{W}_s$.

$$score(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e) = \mathbf{h}_{t-1}^d \mathbf{W}_s \mathbf{h}_j^e \tag{2.75}$$

Bobot W_s, yang kemudian dilatih selama pelatihan *end-to-end* yang normal, memberikan jaringan kemampuan untuk mempelajari aspek kesamaan yang mana antara state *decoder* dan *encoder* yang penting untuk aplikasi saat ini. Model *bilinear* ini juga memungkinkan *encoder* dan *decoder* untuk menggunakan vektor dimensi yang berbeda, sedangkan *dot-product attention* sederhana mensyaratkan bahwa *hidden state encoder* dan *decoder* memiliki dimensi yang sama.

2.3.7.4 Beam Search

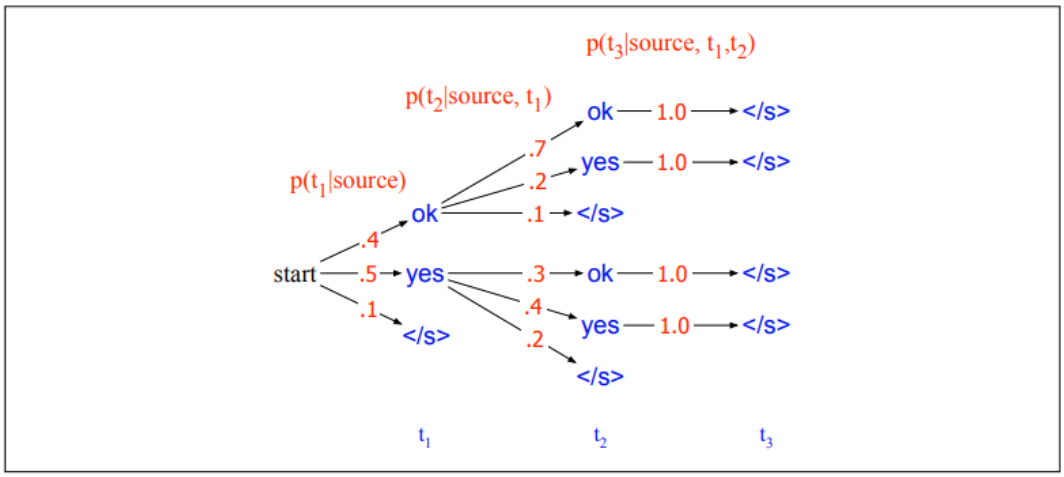
Algoritma *decoding* yang diberikan di atas untuk menghasilkan terjemahan memiliki masalah (seperti halnya *autoregressive generation* yang diperkenalkan di pembahasan sebelumnya untuk menghasilkan dari model bahasa bersyarat). Algoritmanya, pada setiap langkah waktu dalam *decoding*, output y_t dipilih dengan menghitung *softmax* di atas himpunan output yang mungkin (*vocabulary*, dalam kasus pemodelan bahasa atau MT), dan kemudian memilih token dengan probabilitas tertinggi (*argmax*):

$$\hat{y}_t = \text{argmax}_{w \in V} P(w|x, y_1 \ldots y_{t-1}) \tag{2.76}$$

Memilih satu token yang paling mungkin untuk dihasilkan pada setiap langkah disebut *greedy decoding*; algoritma *greedy* adalah salah satu yang membuat pilihan yang optimal secara lokal, apakah itu akan menjadi pilihan terbaik dengan melihat ke belakang atau tidak.

Memang, pencarian *greedy* tidak optimal, dan mungkin tidak menemukan terjemahan probabilitas tertinggi. Masalahnya adalah bahwa token yang terlihat bagus untuk *decoder* sekarang mungkin akan menjadi pilihan yang salah di kemudian hari.

Dibentuk sebuah contoh kasus dengan melihat pohon pencarian, representasi grafis dari pilihan yang dibuat dekoder dalam mencari terjemahan terbaik, di mana kita melihat masalah *decoding* sebagai pencarian ruang keadaan heuristik dan secara sistematis mengeksplorasi ruang kemungkinan keluaran. Dalam pohon pencarian seperti itu, cabang adalah tindakan, dalam hal ini tindakan menghasilkan token, dan node adalah state, dalam hal ini state telah menghasilkan awalan tertentu. Dicarilah urutan tindakan terbaik, yaitu string target dengan probabilitas tertinggi. Gambar 2.3.33 menunjukkan masalah, menggunakan contoh yang dibuat. Perhatikan bahwa urutan yang paling mungkin adalah *ok ok* </s> (dengan probabilitas .4*.7*1.0), tetapi algoritma pencarian *greedy* akan gagal menemukannya, karena salah memilih *yes* sebagai kata pertama karena memiliki probabilitas lokal tertinggi.

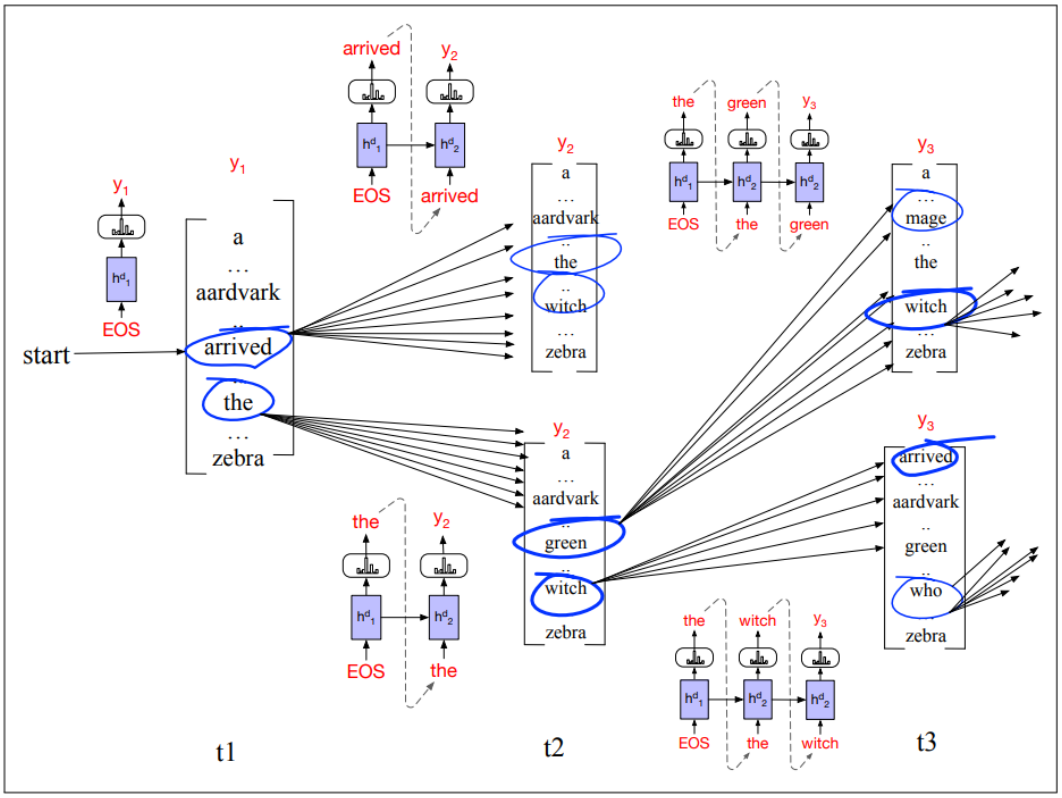


Gambar 2.3.33 Pohon pencarian untuk menghasilkan string target $T = t_1,t_2,...$ dari *vocabulary* V = {yes,ok,<s>}, dengan string sumber, menunjukkan probabilitas menghasilkan setiap token dari state tersebut. Pencarian *greedy* akan memilih *yes* pada langkah pertama diikuti oleh *yes*, daripada urutan yang paling mungkin secara global *ok ok*.

Untuk *tagging part-of-speech* akan menggunakan pencarian pemrograman dinamis (algoritma Viterbi) untuk mengatasi masalah ini. Sayangnya, pemrograman dinamis tidak berlaku untuk masalah *generation* dengan ketergantungan jarak jauh antara keputusan keluaran. Satu-satunya metode yang dijamin untuk menemukan solusi terbaik adalah *exhaustive search*: menghitung probabilitas setiap salah satu dari kemungkinan V^T kalimat (untuk beberapa nilai panjang *T*) yang jelas terlalu lambat.

Sebaliknya, *decoding* di MT dan masalah generasi sequence lainnya umumnya menggunakan metode yang disebut *beam search*. Dalam *beam search*, alih-alih memilih token terbaik untuk dihasilkan pada setiap langkah waktu, akan menyimpan k token yang mungkin pada setiap langkah. Jejak memori ukuran tetap *k* ini disebut *beam width*, pada metafora sinar senter yang dapat diparameterisasikan menjadi lebih lebar atau lebih sempit.

Jadi pada langkah pertama *decoding*, akan menghitung *softmax* atas seluruh *vocabulary*, menetapkan probabilitas untuk setiap kata. Kemudian akan memilih opsi *k*-terbaik dari output *softmax* ini. Keluaran *k* awal ini adalah batas pencarian dan *k* kata awal ini disebut hipotesis. Hipotesis adalah sequence keluaran, terjemahan sejauh ini, bersama dengan probabilitasnya.



Gambar 2.3.34 *Beam search encoding* dengan *beam width* k = 2. Pada setiap langkah waktu, akan dipilih k hipotesis terbaik, menghitung V kemungkinan perluasan dari setiap hipotesis, menilai hasil k*V kemungkinan hipotesis dan memilih k terbaik untuk melanjutkan. Pada waktu 1, perbatasan diisi dengan 2 opsi terbaik dari state awal dekoder: *arrived* dan *the*. Kemudian akan memperluas masing-masing, menghitung probabilitas semua hipotesis sejauh ini (*arrived the*, *arrived aardvark*, *the green*, *the witch*) dan menghitung 2 terbaik (dalam hal ini *the green* dan *the witch*) menjadi perbatasan pencarian untuk memperpanjang pada langkah berikutnya. Pada panah ditunjukkan dekoder yang dijalankan untuk menilai kata ekstensi (untuk kesederhanaannya, belum menunjukkan nilai konteks c_i yang dimasukkan pada setiap langkah).

Pada langkah selanjutnya, masing-masing dari k hipotesis terbaik diperluas secara bertahap dengan diteruskan ke dekoder yang berbeda, yang masing-masing menghasilkan *softmax* di seluruh *vocabulary* untuk memperluas hipotesis ke setiap kemungkinan token berikutnya. Masing-masing hipotesis k*V ini diberi skor oleh $P(y_i | x, y_{<i})$: produk dari peluang pilihan kata saat ini dikalikan dengan peluang jalur yang menuju ke sana. Kemudian akan memangkas hipotesis k*V ke k hipotesis terbaik, sehingga tidak pernah ada lebih dari k hipotesis di perbatasan pencarian, dan tidak pernah lebih dari k dekoder.

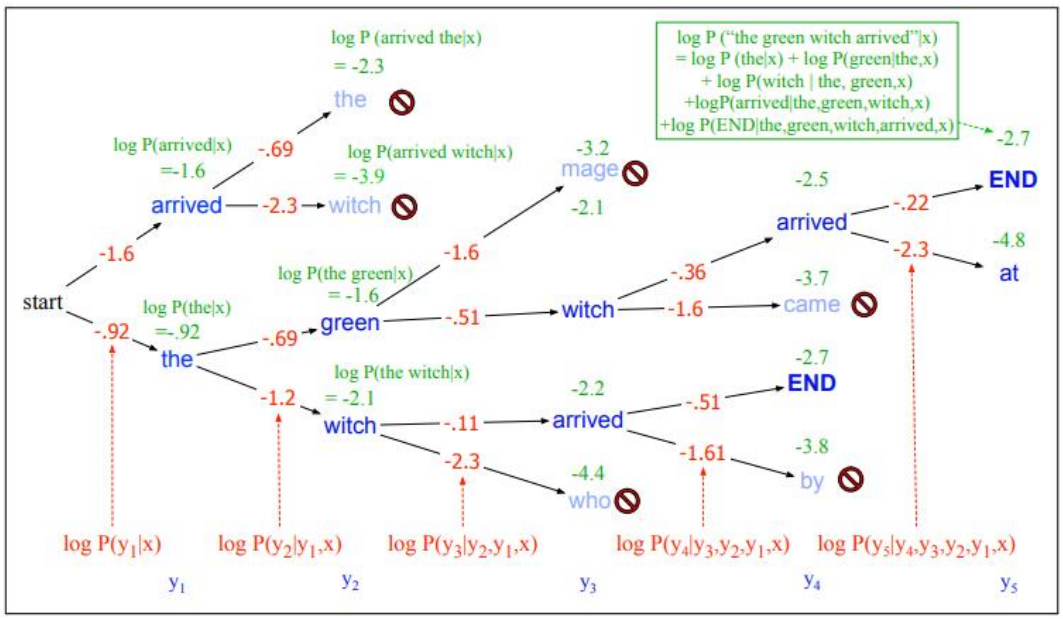
Gambar 2.3.34 mengilustrasikan proses ini dengan *beam width* 2.

Proses ini berlanjut hingga </s> dihasilkan yang menunjukkan bahwa kandidat keluaran lengkap telah ditemukan. Pada titik ini, hipotesis lengkap dihapus dari perbatasan dan ukuran beam dikurangi satu. Pencarian berlanjut hingga beam direduksi menjadi 0. Hasilnya akan menjadi *k* hipotesis.

Untuk bagaimana penilaian bekerja secara detail, menilai setiap node dengan probabilitas lognya. Dapat menggunakan aturan rantai probabilitas untuk memecah $p(y|x)$ menjadi produk dari probabilitas setiap kata yang diberikan konteks sebelumnya, yang dapat diubah menjadi jumlah log (untuk string keluaran dengan panjang t):

$$\begin{aligned} score(y) &= \log P(y|x) \\ &= \log\left(P(y_1|x)P(y_2|y_1,x)P(y_3|y_1,y_2,x)...P(y_t|y_1,...,y_{t-1},x)\right) \\ &= \sum_{i=1}^{t} \log P(y_i|y_1,...,y_{i-1},x) \end{aligned} \tag{2.77}$$

Jadi pada setiap langkah, untuk menghitung probabilitas terjemahan parsial, cukup menambahkan probabilitas log dari terjemahan awalan sejauh ini ke probabilitas log menghasilkan token berikutnya. Gambar 2.3.35 menunjukkan skor untuk contoh kalimat yang ditunjukkan pada Gambar 2.3.34, menggunakan beberapa probabilitas yang dibuat. Probabilitas log akan negatif atau 0, dan maksimum dua probabilitas log adalah yang lebih besar (mendekati 0).



Gambar 2.3.35 Penskoran untuk *decoding beam search* dengan *beam width* k = 2. probabilitas log akan dipertahankan dari setiap hipotesis dalam berkas dengan

menambahkan *logprob* secara bertahap untuk menghasilkan setiap token berikutnya. Hanya *k* jalur teratas yang diperpanjang ke langkah berikutnya.

Satu masalah muncul dari fakta bahwa hipotesis lengkap mungkin memiliki panjang yang berbeda. Karena model umumnya menetapkan probabilitas yang lebih rendah untuk string yang lebih panjang, algoritma *naive* juga akan memilih string yang lebih pendek untuk *y*. Ini tidak menjadi masalah selama langkah-langkah decoding sebelumnya; karena *beam search* yang bersifat *breadth-first* semua hipotesis yang dibandingkan memiliki panjang yang sama. Solusi yang biasa untuk ini adalah dengan menerapkan beberapa bentuk normalisasi panjang untuk masing-masing hipotesis, misalnya hanya membagi probabilitas log negatif dengan jumlah kata:

$$score(y) = -\log P(y|x) = \frac{1}{T}\sum_{i=1}^{t} -\log P(y_i|y_1,...,y_{i-1},x) \quad (2.78)$$

Beam search umum digunakan dalam sistem MT produksi besar, umumnya dengan *beam width k* antara 5 dan 10. Apa yang akan dilakukan dengan hipotesis *k* yang dihasilkan? Dalam beberapa kasus, yang dibutuhkan dari algoritma MT adalah hipotesis tunggal terbaik, sehingga dapat mengembalikannya. Dalam kasus lain, aplikasi *downstream* mungkin akan melihat semua *k* hipotesis, sehingga kami dapat meneruskan semuanya (atau subset) ke aplikasi *downstream* dengan skor masing-masing.

2.3.7.5 Detil Praktisi Dalam Membangun Sistem MT

2.3.7.5.1 Tokenisasi

Sistem terjemahan mesin umumnya menggunakan *vocabulary* tetap. Cara umum untuk menghasilkan *vocabulary* ini adalah dengan algoritma *BPE* atau *wordpiece*. Umumnya *vocabulary* bersama digunakan untuk bahasa sumber dan bahasa target, yang memudahkan untuk menyalin token (seperti nama) dari sumber ke target, jadi akan dibangun leksikon wordpiece/BPE pada korpus yang berisi data bahasa sumber dan bahasa target. Wordpieces menggunakan simbol khusus di awal setiap token; inilah tokenisasi yang dihasilkan dari sistem *Google MT* (Wu et al., 2016):

words: Jet makers feud over seat width with big orders at stake
wordpieces: _J et _makers _fe ud _over _seat _width _with _big _orders _at _stake

Gambar 2.3.36 Perbedaan bentuk kata dan *wordpieces*

Alih-alih mendefinisikan token sebagai kata (apakah dibatasi oleh spasi atau algoritma yang lebih kompleks), atau sebagai karakter (seperti dalam bahasa Cina), dapat digunakan data untuk secara otomatis memberi tahu bagaimana seharusnya token itu. Ini sangat berguna dalam menangani kata-kata yang tidak dikenal, masalah penting dalam pemrosesan bahasa. Algoritma NLP sering mempelajari beberapa fakta tentang bahasa dari satu korpus (korpus pelatihan) dan kemudian menggunakan fakta ini untuk membuat keputusan tentang korpus uji terpisah dan bahasanya. Jadi jika korpus pelatihan ini berisikan, seperti *low*, *new*, *newer*, tetapi tidak *lower*, maka jika kata *lower* muncul di korpus pengujiannya, sistem tidak akan tahu apa yang harus dilakukan dengannya.

Untuk mengatasi masalah *unknown word problem* ini, *tokenizer* modern sering kali secara otomatis menginduksi set token yang menyertakan token yang lebih kecil dari kata, yang disebut *subwords*. *Subwords* dapat berupa *substring* acak, atau dapat berupa unit yang mengandung makna seperti morfem *-est* atau *-er*. (Sebuah morfem adalah unit pembawa makna terkecil dari suatu bahasa; misalnya kata *unlikeliest* memiliki morfem *un-*, *likely*, dan *-est.*) Dalam skema tokenisasi modern, sebagian besar token adalah kata, tetapi beberapa token sering menjadi morfem atau *subwords* lain seperti *-er*. Setiap kata yang tidak terlihat seperti *lower* dengan demikian dapat diwakili oleh beberapa sequence unit *subword* yang diketahui, seperti *low* dan *er*, atau bahkan sebagai sequence huruf individual jika diperlukan.

Kebanyakan skema *tokenization* memiliki dua bagian: *token learner*, dan *token segmenter*. *Token learner* mengambil korpus pelatihan mentah (kadang-kadang secara kasar dipisahkan menjadi kata-kata, misalnya dengan spasi) dan menginduksi *vocabulary*, satu set token. *Token segmenter* mengambil kalimat uji mentah dan mengelompokkannya ke dalam token dalam *vocabulary*. Tiga algoritma banyak digunakan: *byte-pair encoding* (Sennrich et al., 2016a), *unigram language modeling* (Kudo, 2018), dan *WordPiece* (Schuster & Nakajima, 2012);

ada juga *library SentencePiece* yang mencakup implementasi dua dari tiga yang pertama (Kudo & Richardson, 2018).

Berikut adalah rincian lebih lanjut tentang algoritma *wordpiece*, yang diberikan korpus pelatihan dan ukuran *vocabulary* yang diinginkan V, dan hasil sebagai berikut:

1. Inisialisasi leksikon kata dengan karakter (misalnya subset karakter Unicode, menciutkan semua karakter yang tersisa ke token karakter khusus yang tidak diketahui).
2. Ulangi sampai ada V *wordpieces*:
    a. Latih model bahasa *n-gram* pada korpus pelatihan, menggunakan kumpulan *wordpieces* saat ini.
    b. Pertimbangkan himpunan kemungkinan *wordpieces* baru yang dibuat dengan menggabungkan dua kata dari leksikon saat ini. Pilih satu *wordpieces* baru yang paling meningkatkan kemungkinan model bahasa dari korpus pelatihan.

Vocabulary dari 8K hingga 32K *wordpieces* biasanya digunakan.

2.3.7.5.2 Sentencepiece

SentencePiece (Kudo & Richardson, 2018) adalah *tokenizer* dan *detokenizer* teks *unsupervised* terutama untuk sistem pembuatan teks berbasis *Neural Network* di mana ukuran *vocabulary* telah ditentukan sebelumnya sebelum pelatihan model *neural*. SentencePiece mengimplementasikan unit *subword* (misalnya, *byte-pair-encoding* (BPE) (Sennrich et al., 2016a)) dan model bahasa *unigram* (Kudo, 2018)) dengan perluasan pelatihan langsung dari kalimat mentah. *Sentencepiece* memungkinkan untuk membuat sistem *end-to-end* murni yang tidak bergantung pada pre/post-*processing* khusus bahasa.

Jumlah token unik telah ditentukan sebelumnya di *Sentencepiece*. Model NMT biasanya beroperasi dengan *vocabulary* tetap. Tidak seperti kebanyakan algoritma segmentasi kata *unsupervised*, yang mengasumsikan *vocabulary* tak terbatas, *Sentencepiece* melatih model segmentasi sedemikian rupa sehingga ukuran *vocabulary* akhir tetap, misalnya, 8k, 16k, atau 32k.

Perhatikan bahwa *Sentencepiece* menentukan ukuran *vocabulary* akhir untuk pelatihan, yang berbeda dari *subword-nmt* yang menggunakan jumlah operasi gabungan. Jumlah operasi penggabungan adalah parameter khusus BPE dan tidak berlaku untuk algoritma segmentasi lainnya, termasuk *unigram*, kata, dan karakter.

Implementasi *sub-word* sebelumnya mengasumsikan bahwa kalimat input adalah pre-*tokenized*. Batasan ini diperlukan untuk pelatihan yang efisien, tetapi membuat *preprocessing* menjadi rumit karena harus menjalankan *tokenizer* yang bergantung pada bahasa terlebih dahulu. Implementasi *Sentencepiece* cukup cepat untuk melatih model dari kalimat mentah. Ini berguna untuk melatih *tokenizer* dan *detokenizer* untuk bahasa Cina dan Jepang di mana tidak ada spasi eksplisit di antara kata-kata.

Langkah pertama pemrosesan Bahasa Alami adalah *tokenisasi* teks. Misalnya, tokenizer bahasa Inggris standar akan mengelompokkan teks " Hello world. " menjadi tiga token seperti berikut.

```
[Hello] [World] [.]
```

Pada suatu pengamatan, bahwa input asli dan urutan tokenized tidak dapat dikonversi secara *reversible*. Misalnya, informasi yang tidak ada spasi antara "World" dan "." dijatuhkan dari *sequence tokenized*, karena misalnya,

```
Tokenize(“World.”) == Tokenize(“World .”)
```

Sentencepiece memperlakukan teks *input* hanya sebagai *sequence* karakter *Unicode*. Spasi juga ditangani sebagai simbol normal. Untuk menangani *whitespace* sebagai token dasar secara eksplisit, *Sentencepiece* pertama-tama mengeluarkan *whitespace* dengan simbol meta " " (U+2581) sebagai berikut.

```
Hello▁World.
```

Kemudian, teks ini tersegmentasi menjadi potongan-potongan kecil, misalnya:

```
[Hello] [▁Wor] [ld] [.]
```

Karena *whitespace* dipertahankan dalam teks tersegmentasi, dapat dilakukan *detokenization* teks tanpa ambiguitas.

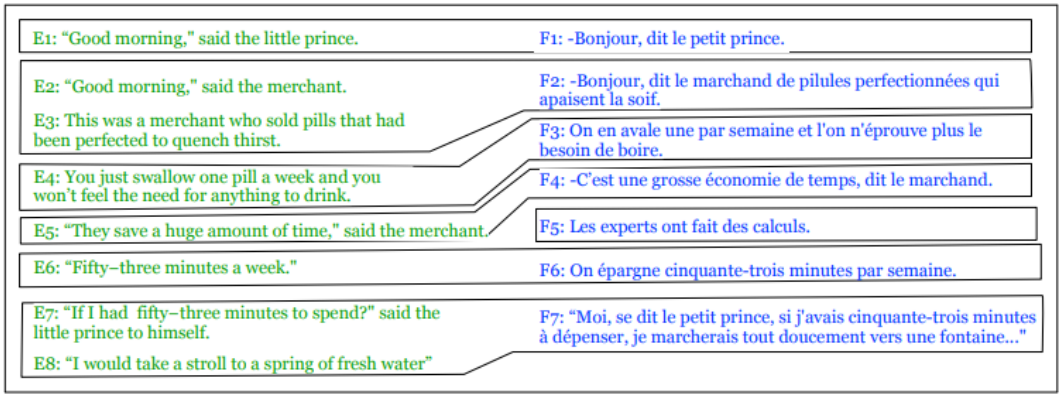
```
detokenized = ''.join(pieces).replace('▁', ' ')
```

Fitur ini memungkinkan untuk melakukan *detokenization* tanpa bergantung pada sumber daya khusus bahasa.

2.3.7.5.3 Korpus MT

Model terjemahan mesin dilatih pada korpus paralel, kadang-kadang disebut *biteks*, teks yang muncul dalam dua (atau lebih) bahasa. Sejumlah besar korpora paralel tersedia. Beberapa adalah sumber pemerintah; korpus *Europarl* (Koehn, 2005), diambil dari proceedings Parlemen Eropa, berisi antara 400.000 dan 2 juta kalimat masing-masing dari 21 bahasa Eropa. Korpus Paralel Perserikatan Bangsa-Bangsa berisi urutan 10 juta kalimat dalam enam bahasa resmi Perserikatan Bangsa-Bangsa (Arab, Cina, Inggris, Prancis, Rusia, Spanyol) (Marcin and Pouliquen, 2016). Korpus paralel lainnya telah dibuat dari subtitel film dan TV, seperti korpus *OpenSubtitles* (Lison & Tiedemann, 2016), atau dari teks web umum, seperti korpus *ParaCrawl* dari 223 juta pasangan kalimat antara 23 bahasa EU dan bahasa Inggris yang diekstraksi dari *CommonCrawl* (Bañón et al., 2020).

Korpus pelatihan standar untuk MT hadir sebagai pasangan kalimat yang selaras. Saat membuat corpora baru, misalnya untuk bahasa yang kurang sumber daya atau domain baru, perataan kalimat ini harus dibuat. Gambar 2.3.37 memberikan contoh keselarasan kalimat hipotetis.



Gambar 2.3.37 Contoh penyelarasan antara kalimat dalam bahasa Inggris dan Prancis, dengan kalimat yang diambil dari *Le Petit Prince* karya Antoine de Saint-Exupery dan terjemahan hipotetis.

Penjajaran kalimat pada gambar diatas mengambil kalimat e_1,..., e_n, dan f_1,..., f_n dan menemukan himpunan minimal kalimat yang merupakan terjemahan satu sama lain, termasuk pemetaan kalimat tunggal seperti (e_1,f_1), (e_4,f_3), (e_5,f_4), (e_6,f_6) serta alignment 2-1 (e_2/e_3,f_2), (e_7/e_8,f_7), dan alignment null (f_5).

2.3.7.6 Evaluasi *Language Models*

Cara terbaik untuk mengevaluasi kinerja model bahasa adalah dengan menyematkannya dalam aplikasi dan mengukur seberapa banyak peningkatan dari aplikasinya. Evaluasi *end-to-end* seperti itu disebut evaluasi ekstrinsik. Evaluasi ekstrinsik adalah satu-satunya cara untuk mengetahui apakah peningkatan tertentu dalam suatu komponen benar-benar akan membantu tugas tugas yang ada. Jadi, untuk pengenalan suara, dapat dibandingkan kinerja dua model bahasa dengan menjalankan pengenal suara dua kali, sekali dengan masing-masing model bahasa, dan melihat mana yang memberikan transkripsi yang lebih akurat.

Sayangnya, menjalankan sistem NLP besar secara end-to-end seringkali sangat mahal. Sebaliknya, akan lebih baik jika memiliki metrik yang dapat digunakan untuk mengevaluasi potensi peningkatan dalam model bahasa dengan cepat. Metrik evaluasi intrinsik adalah metrik yang mengukur kualitas model yang tidak bergantung pada aplikasi apa pun.

Untuk evaluasi intrinsik dari model bahasa kita membutuhkan satu *test set*. Seperti banyak model statistik lainnya, probabilitas model n-gram berasal dari korpus yang dilatihnya, *training set* atau korpus pelatihan. Kemudian dapat diukur kualitas model n-gram dengan kinerjanya pada beberapa data yang tidak terlihat yang disebut *test set* atau korpus *test*.

Jadi jika diberikan korpus teks dan ingin membandingkan dua model n-gram yang berbeda, akan dibagi data-nya menjadi set pelatihan dan pengujian, melatih parameter kedua model pada set pelatihan, dan kemudian membandingkan seberapa baik kedua model yang dilatih. sesuai dengan set *test*.

Tapi apa artinya "sesuai dengan set *test*"? Hal tersebut dapat merupakan model mana pun yang memberikan probabilitas lebih tinggi pada set *test*-nya, artinya model tersebut memprediksi set pengujian tersebut dengan lebih akurat menjadikannya model yang lebih baik. Diberikan dua model probabilistik, model yang lebih baik adalah model yang memiliki kecocokan yang lebih ketat dengan data *test* atau yang lebih baik memprediksi detail data *test*, dan karenanya akan menetapkan probabilitas yang lebih tinggi untuk data *uji*.

Karena metrik evaluasi didasarkan pada probabilitas set *test*, penting untuk tidak memasukkan kalimat *test set* tersebut ke dalam set pelatihan. Misalkan untuk

menghitung probabilitas kalimat "tes" tertentu. Jika kalimat pengujian(test) adalah bagian dari korpus pelatihan(training), akan terlihat keliru saat menetapkannya dengan nilai probabilitas tinggi ketika itu terjadi di set *test*. Hal tersebut dapat dikatakan sebagai situasi "*training on the test set*". Pelatihan pada set test memperkenalkan bias yang membuat semua probabilitas terlihat terlalu tinggi, dan menyebabkan ketidakakuratan besar dalam *perplexity*, metrik berbasis probabilitas.

Kadang-kadang ketika menggunakan set tes tertentu begitu sering sehingga secara implisit menyesuaikan dengan karakteristiknya. Muncul kebutuhan set *test* baru yang benar-benar tidak terlihat. Dalam kasus seperti itu, terdapat set pengujian awal yang dikenal sebagai *development test set* atau, *devset*. Bagaimana cara membagi data ke dalam set pelatihan(*training*), pengembangan(*development*), dan pengujian(*test*)? Set *test* dapat dibentuk menjadi sebesar mungkin, karena set *test* yang kecil mungkin secara tidak sengaja tidak representatif, tetapi juga dibutuhkan sebanyak mungkin data pelatihan. Minimal, untuk memilih set *test* terkecil yang memberi kekuatan statistik yang cukup untuk mengukur perbedaan yang signifikan secara statistik antara dua model potensial. Dalam praktiknya, hanya membagi data tersebut menjadi 80% pelatihan(*training*), 10% pengembangan(*development*), dan 10% uji(*test*). Mengingat korpus besar yang ingin dibagi menjadi pelatihan dan pengujian, data *test* dapat diambil dari beberapa sequence teks yang berkelanjutan di dalam korpus, atau dapat menghapus "garis" teks yang lebih kecil dari bagian yang dipilih secara acak dari korpus dan menggabungkannya ke dalam satu set *test*.

2.3.7.7 Evaluasi MT

Hasil terjemahan dalam MT dievaluasi dengan dua dimensi berikut:

1. Adequacy: seberapa baik terjemahan menangkap makna yang tepat dari kalimat sumber. Kadang-kadang disebut *faithfulness* atau *fidelity*.
2. Fluency: seberapa lancar terjemahan dalam bahasa sasaran (apakah gramatikal, jelas, mudah dibaca, alami).

Menggunakan manusia untuk mengevaluasi adalah yang paling akurat, tetapi metrik otomatis juga digunakan untuk kenyamanan.

2.3.7.7.1 Penilaian Berbasis Manusia

Evaluasi yang paling akurat menggunakan penilai manusia, seperti *crowdworkers online*, untuk mengevaluasi setiap terjemahan di sepanjang dua dimensi. Misalnya, sepanjang dimensi *fluency*, kita dapat menanyakan seberapa dapat untuk dimengerti, seberapa jelas, seberapa mudah dibaca, atau seberapa natural output MT (teks target). Kita dapat memberi penilai skala, misalnya, dari 1 (benar-benar tidak dapat dipahami) hingga 5 (benar-benar dapat dipahami, atau 1 hingga 100, dan meminta mereka untuk menilai setiap kalimat atau paragraf dari keluaran MT.

Hal yang sama dapat dilakukan untuk menilai dimensi kedua, *adequacy*, menggunakan penilai untuk menetapkan skor pada skala. Jika memiliki penilai bilingual, dengan memberi mereka kalimat sumber dan kalimat target yang diusulkan, dan menilai, pada skala 5 poin atau 100 poin, berapa banyak informasi dalam sumber yang dipertahankan dalam target. Jika hanya memiliki penilai monolingual tetapi memiliki terjemahan manusia yang baik dari teks sumber, dapat dilakukan dengan memberikan penilai monolingual terjemahan referensi manusia dan terjemahan mesin target dan sekali lagi menilai seberapa banyak informasi yang disimpan. Alternatifnya adalah dengan melakukan *ranking*: beri penilai sepasang kandidat terjemahan, dan tanyakan mana yang mereka sukai.

Pelatihan penilai manusia (yang sering kali merupakan *crowdworker online*) sangat penting; penilai tanpa keahlian penerjemahan merasa sulit untuk memisahkan *fluency* dan *adequacy*, sehingga pelatihan menyertakan contoh yang membedakannya dengan cermat. Penilai sering tidak setuju (kalimat sumber mungkin ambigu, penilai akan memiliki pengetahuan dunia yang berbeda, penilai mungkin menerapkan skala secara berbeda). Oleh karena itu adalah umum untuk menghapus penilai *outlier*, dan (jika kita menggunakan skala yang cukup halus) menormalkan penilai dengan mengurangi rata-rata dari skor mereka dan membaginya dengan varians.

2.3.7.7.2 Penilaian Otomatis

Sementara manusia menghasilkan evaluasi terbaik dari keluaran terjemahan mesin, menjalankan evaluasi manusia dapat memakan waktu dan mahal.

Untuk alasan ini metrik otomatis sering digunakan. Metrik otomatis kurang akurat dibandingkan evaluasi manusia, tetapi dapat membantu menguji potensi peningkatan sistem, dan bahkan digunakan sebagai fungsi *loss* otomatis untuk pelatihan.

2.3.7.8 BLEU

Bilingual Evaluation UnderStudy atau disingkat menjadi BLEU adalah suatu metode yang dapat melakukan evaluasi mengenai kualitas hasil terjemahan suatu mesin penerjemah dari bahasa satu ke bahasa lainnya. BLEU bekerja dengan cara mengukur skor presisi dari modified n-gram dari hasil terjemahan prediksi dengan terjemahan aslinya serta menggunakan konstanta brevity penalty. Rumus BLEU ditunjukkan pada persamaan (1), (2), dan (3) berikut:

$$BP_{BLEU} = \begin{cases} 1 \\ e^{1-\frac{r}{c}} \end{cases} \begin{matrix} if\ c > r \\ c \leq r \end{matrix} \quad \text{.....(2.79)}$$

$$P_n = \frac{\sum C \in corpus\ n-gram \in c \sum count_{clip}(n-gram)}{\sum C \in corpus\ n-gram \in c \sum count(n-gram)} \quad \text{.....(2.80)}$$

$$BLEU = BP_{BLEU}.\ exp^{\sum_{n-1}^{N} W_n log P_n} \quad \text{.....(2.81)}$$

Keterangan:

BP	=	*brevity penalty*
c	=	jumlah kata dari terjemahan otomatis
r	=	jumlah kata rujukan
P_n	=	*modified precission score*
W_n	=	1/N (standar nilai N untuk BLEU adalah 4)

2.4 MarianNMT

Sebuah toolkit Neural Machine Translation yang dikembangkan oleh tim *Microsoft Translator* dengan harapan menciptakan *toolkit* yang *resource-friendly* dan dapat mencapai kecepatan training dan terjemahan yang tinggi(Junczys-Dowmunt et al., 2018).

Marian dikembangkan dengan dasar implementasi ulang dari *Nematus* (Sennrich et al., 2017) menggunakan bahasa pemograman C++11 secara menyeluruh, yang membuat *toolkit* ini dikhususkan untuk implementasi efektif

dalam bahasa diatas saja. Marian juga menggunakan *back-end deep learning* yang didasarkan pada *reverse-mode auto-differentiation with dynamic computation graphs*, dimana *back-end* dikhususkan untuk optimasi kinerja *machine translation*.

Kebutuhan implementasi Marian tidak besar, hanya menggunakan *library Boost*, dan *CUDA* atau *BLAS*, bahkan bisa keduanya. Untuk library *Boost* sendiri akan direncanakan untuk *deprecated* bertahap. Jalur optimasi yang ditawarkan cukup banyak, seperti *MPI-based multi-node training*, *efficient batched beam search*, *compact new model implementation* (*encoder-decoder* dengan tipe berbeda), *custom operator*, dan *custom GPU kernel*. *CPU backend* bahkan menerima kontribusi dan dioptimasi lanjut oleh perusahaan microprocessor terkemuka *Intel*.

Dalam ujicoba studi kasusnya, Marian digunakan untuk memfasilitasi beberapa riset NLP yang sudah ada. Seperti pada riset nilai BLEU tertinggi untuk terjemahan berita bahasa Inggris-Jerman dalam WMT2017 *shared task*. Dalam hasilnya Marian mampu mereplikasikan hasil dari penelitian tersebut secara kualitas, disertai kecepatan training 4 kali lebih cepat di GPU tunggal dari *framework* sebelumnya (*Nematus*, sekitar 2800 w/s), bahkan mencapai 30 kali lebih cepat ketika menggunakan 8 GPU identik. Di terjemahannya, untuk *back-translation* sebanyak 10M kalimat dengan model dasar butuh waktu 4 jam di 8 GPU dengan kecepatan sekitar 15.850 tok/s, dan dengan rumus sama (*beam size* 5, *batch size* 64) ke beberapa *batch size* berbeda dengan GPU tunggal.

Pada submisi terbarunya di tahun 2019, MarianNMT di versi terbarunya, dengan berbagai modifikasi terhadap sistemnya mampu membawa kinerjanya untuk lebih cepat 24x untuk CPU-build dan 14x untuk GPU-build dari submisi sebelumnya di tahun 2018. (Kim et al., 2019)

Tabel 2.4.1 Waktu translasi dalam detik untuk newstest-2017 (3,004 kalimat, 76,501 source BPE tokens) untuk arsitektur dan batch size berbeda. (Junczys-Dowmunt et al., 2018)

Model	1	8	64
Shallow RNN	112.3	236.5	15.7
Deep Transition RNN	179.4	36.5	21.0
Transformer	362.7	98.5	71.3

2.4.1 Arsitektur Neural Network MarianNMT

MarianNMT mengikuti dasar arsitektur *encoder-decoder* dengan *attention* (Bahdanau et al., 2015) dengan beberapa perbedaan implementasi.

- Inisialisasi *decoder hidden state* dengan *mean* dari *source annotation*, tidak dengan *annotation* di akhir posisi dari *encoder backward* RNN.
- *Conditional GRU* dengan mekanisme *attention.*
- Dalam *decoder*, digunakan *feedforward hidden layer* dengan fungsi *non-linearty tanh*, menggantikan *maxout* sebelum *softmax layer*
- Dalam *embedding layer* untuk kata di *encoder-decoder*, *bias* tidak digunakan
- Menggunakan fase *decoder 'Look, Update, Generate'*. Model (Bahdanau et al., 2015) pada umumnya menggunakan *'Look, Generate, Update'*.
- *Bayesian dropout* (Gal & Ghahramani, 2015) yang optional
- Representasi input yang mendukung *multiple features* (atau faktor) pada tiap *time step*, dengan *final embedding* menjadi pemersatu tiap *embedding* di *feature*, daripada dengan *word embedding* tunggal pada tiap posisi asal. (Sennrich & Haddow, 2016)
- *Tying embedding* untuk matriks (Inan et al., 2016; Press & Wolf, 2016)

Didapat sebuah source sequence ($x_1,\ldots,x_{Tx}$) dari panjang T_x dan target sequence ($y_1,\ldots,y_{Ty}$) dari panjang T_y, h_i akan dijadikan *annotation* dari *source symbol* pada posisi *i*, didapat dengan menggabungkan *forward* dan *backward encoder* dari RNN *hidden state*, $h_i = \left[\overrightarrow{h_i};\overleftarrow{h_i}\right]$, dan s_j akan menjadi *decoder hidden state* di posisi *j*.

RNNSearch (Bahdanau et al., 2015)		Nematus (DL4MT)	
Phase	Output - Input	Phase	Output - Input
Look	$\mathbf{c}_j \leftarrow \mathbf{s}_{j-1}, \mathrm{C}$	Look	$\mathbf{c}_j \leftarrow \mathbf{s}_{j-1}, y_{j-1}, \mathrm{C}$
Generate	$y_j \leftarrow \mathbf{s}_{j-1}, y_{j-1}, \mathbf{c}_j$	Update	$\mathbf{s}_j \leftarrow \mathbf{s}_{j-1}, y_{j-1}, \mathbf{c}_j$
Update	$\mathbf{s}_j \leftarrow \mathbf{s}_{j-1}, y_j, \mathbf{c}_j$	Generate	$y_j \leftarrow \mathbf{s}_j, y_{j-1}, \mathbf{c}_j$

Gambar 2.4.1 Perbandingan RNNSearch & Nematus untuk algoritma *phase decoder*

(Bahdanau et al., 2015) melakukan inisialisasi *decoder hidden state s*

dengan *encoder state* yang membelakangi di posisi terakhir.

$$\mathbf{s}_0 = \tanh\left(\mathbf{W}_{init}\overleftarrow{\mathbf{h}}_1\right) \tag{2.82}$$

Dengan W_{init} sebagai *trained parameter*, Marian ditentukan menggunakan *annotation* rerata.

$$\mathbf{s}_0 = \tanh\left(\mathbf{W}_{init}\frac{\sum_{i=1}^{T_x}\mathbf{h}_i}{T_x}\right) \tag{2.83}$$

Marian mengimplementasikan *conditional GRU with attention*, cGRU_{att}. Bekerja dengan menggunakan *hidden state* sebelumnya s_{j-1}, seluruh set sumber anotasi C = $\{h_1,\ldots,h_{Tx}\}$ dan simbol terkode sebelumnya y_{j-1} berurutan untuk memperbarui *hidden state* s_j, yang digunakan selanjutnya untuk mengkodekan symbol y_j di posisi *j*.

$$\mathbf{s}_j = \text{cGRU}_{\text{att}}\left(\mathbf{s}_{j-1}, y_{j-1}, \mathbf{C}\right) \tag{2.84}$$

cGRUatt terdiri atas 3 komponen. Dua blok GRU *state transition* dan sebuah mekanisme *attention* ATT di antaranya. *Transition block* pertama, GRU1 menggabungkan simbol terkodekan sebelumnya y_{j-1} dan hidden state s_{j-1} berurutan untuk menghasilkan representasi intermediate s'_j yang dirumuskan seperti berikut.

$$\begin{aligned}
\mathbf{s}'_j &= \text{GRU}_1\left(y_{j-1}, \mathbf{s}_{j-1}\right) = \left(1-\mathbf{z}'_j\right)\odot\underline{\mathbf{s}}'_j + \mathbf{z}'_j\odot\mathbf{s}_{j-1},\\
\underline{\mathbf{s}}'_j &= \tanh\left(\mathbf{W}'\mathbf{E}[y_{j-1}] + \mathbf{r}'_j\odot\left(\mathbf{U}'\mathbf{s}_{j-1}\right)\right),\\
\mathbf{r}'_j &= \sigma\left(\mathbf{W}'_r\mathbf{E}[y_{j-1}] + \mathbf{U}'_r\mathbf{s}_{j-1}\right),\\
\mathbf{z}'_j &= \sigma\left(\mathbf{W}'_z\mathbf{E}[y_{j-1}] + \mathbf{U}'_z\mathbf{s}_{j-1}\right),
\end{aligned} \tag{2.85}$$

Dimana **E** adalah matriks word embedding target, s'_j adalah representasi *proposal intermediate*, r'_j dan z'_j sebagai *reset* dan *update* dari *gate activation*. Dalam persamaan ini, $W', U', W'_r, U'_r, W'_z, U'_z$ adalah *trained* model parameter, σ adalah fungsi aktivasi *logistic sigmoid*.

Mekanisme attention, ATT akan memasukkan seluruh *context set C* beserta *intermediate hidden state* s'_j berurutan untuk menghitung vektor konteks $\mathbf{c}_j$ sebagai berikut:

$$\begin{aligned}\mathbf{c}_j =& \text{ATT}\left(\mathbf{C}, \mathbf{s}'_j\right) = \sum_i^{T_x} \alpha_{ij}\mathbf{h}_i, \\ \alpha_{ij} =& \frac{\exp(e_{ij})}{\sum_{k=1}^{Tx}\exp(e_{kj})}, \\ e_{ij} =& \mathbf{v}_a^{\intercal}\tanh\left(\mathbf{U}_a\mathbf{s}'_j + \mathbf{W}_a\mathbf{h}_i\right),\end{aligned} \tag{2.86}$$

Dimana α_{ij} adalah *normalized alignment weight* diantara *source symbol* di posisi i dan *target symbol* di posisi j dan v_a, U_a, W_a adalah *trained model parameter*.

Terakhir, *block transition* kedua, GRU2, akan menghasilkan $\mathbf{s}_j$, hidden state dari cGRU_{att}, dengan melihat pada representasi intermediate s'_j dan context vector $\mathbf{c}_j$ dengan persamaan berikut

$$\begin{aligned}\mathbf{s}_j = \text{GRU}_2\left(\mathbf{s}'_j, \mathbf{c}_j\right) &= (1-\mathbf{z}_j)\odot\underline{\mathbf{s}}_j + \mathbf{z}_j\odot\mathbf{s}'_j, \\ \underline{\mathbf{s}}_j &= \tanh\left(\mathbf{W}\mathbf{c}_j + \mathbf{r}_j\odot\left(\mathbf{U}\mathbf{s}'_j\right)\right), \\ \mathbf{r}_j &= \sigma\left(\mathbf{W}_r\mathbf{c}_j + \mathbf{U}_r\mathbf{s}'_j\right), \\ \mathbf{z}_j &= \sigma\left(\mathbf{W}_z\mathbf{c}_j + \mathbf{U}_z\mathbf{s}'_j\right),\end{aligned} \tag{2.87}$$

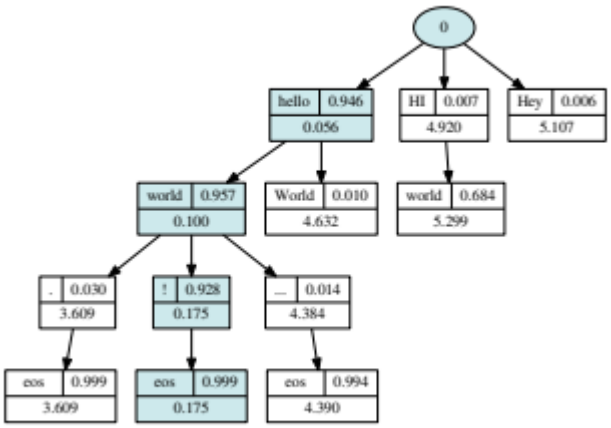
$\underline{s}_j$ memilki peran yang sama menjadi *proposal hidden state*, r'_j dan z'_j sebagai *reset* dan *update* untuk *gate activation* dengan *trained model parameter* $W', U', W'_r, U'_r, W'_z, U'_z$.

Sebagai catatan, kedua blok GRU tidaklah *reccurent* secara sendirinya, melainkan terjadi pada tingkat keseluruhan cGRU layer. Cara kombinasi blok RNN seperti ini mirip dengan literatur *deep transition* RNN (Pascanu et al., 2013; Zilly et al., 2016) sebagai pembaruan dari stacked RNN umum di (Schmidhuber, 1992; Hihi & Bengio, 1995; Graves, 2013).

Deep output, tersebut s_j, y_{j-1}, dan c_j, probabilitas output $p(y_j|s_j, y_{j-1}, c_j)$ yang dihitung oleh *softmax activation*, menggunakan representasi intermediate t_j.

$$\begin{aligned}p(y_j|\mathbf{s}_j, y_{j-1}, \mathbf{c}_j) &= \text{softmax}\left(\mathbf{t}_j\mathbf{W}_o\right) \\ \mathbf{t}_j &= \tanh\left(\mathbf{s}_j\mathbf{W}_{t1} + \mathbf{E}[y_{j-1}]\mathbf{W}_{t2} + \mathbf{c}_j\mathbf{W}_{t3}\right)\end{aligned} \tag{2.88}$$

$W_{t1}, W_{t2}, W_{t3}, W_o$ adalah *trained model parameter*.



Gambar 2.4.2 Visualisasi grafis untuk pencarian pada penerjemahan bahasa DE→ EN "Hallo Welt!" dengan *beam size* 3.

2.4.2 Algoritma Training MarianNMT

Dasarnya, MarianNMT menggunakan *cross-entropy minimization* terhadap korpus training paralel. *Training* akan dilakukan melalui *stochastic gradient descent* atau varian lainnya dengan *adaptive learning rate* (Adadelta (Zeiler, 2012), RmsProp (Tieleman et al., 2012), Adam (Kingma & Ba, 2014)).

Sebagai tambahan, Marian mendukung *minimum risk training* (MRT) (Shen et al., 2016) untuk mengoptimasi secara otomatis, *sentence-level loss function*. Banyak ukuran metrik MT yang didukung sebagai fungsi *loss*, seperti *smoothed sentence-level* BLEU (Chen & Cherry, 2014), METEOR (Denkowski & Lavie, 2011), BEER (Stanojevic & Sima'an, 2014), dan penambahan metrik terimplementasi lainnya.

Untuk menstabilkan training, Marian mendukung *early stopping* yang berdasar *cross entropy* atau fungsi *loss* lainnya yang ditetapkan user.

2.5 FLORES-101

(Goyal et al., 2021) mengenalkan *evaluation benchmark open-source* bernama FLoRes (*Facebook Low Resource*) yang memiliki konsentrasi terhadap 101 bahasa di tiap belah dunia termasuk kawasan Asia, yang kebetulan memiliki beberapa permasalahan dengan validasi-nya. Basis *low-resource* yang ditawarkan memiliki makna bahwa ada beberapa bahasa yang kebetulan memiliki data mentah yang sedikit dan dalam bentuk tidak umum.

Evaluation benchmark ini dapat digunakan dengan sistem *many-to-many*, dan tak terkecuali untuk kasus selain itu. Evaluasi dapat dilakukan secara *local* atau melalui *online benchmark service* DynaBench, yang mana korpus test untuk evaluasi akan diproses secara langsung disana saat menggunakan layanan tersebut. DynaBench sendiri akan merekam hasil skor training dan terjemahan model untuk dipublikasikan disana.

FLORES-101 benchmark memiliki 3001 kalimat yang diambil dari Wikipedia dalam Bahasa inggris dan diterjemahkan kedalam 101 bahasa. Kalimat berukuran sekitar 20 kata, yang berasal dari 1175 artikel berbeda pada 3 domain: WikiNews, WikiJunior, dan WikiVoyage. Tiap artikel rerata memiliki 3 kalimat yang dipilih dari tiap dokumen, dan dokumen tersebut dibagi menjadi dev, devtest dan test set.

Number of Sentences		3001
Average Words per Sentence		21
Number of Articles		842
Average Number of Sentences per Article		3.5
% of Articles with Hyperlinked Entities		40
% of Articles with Images		66
Evaluation Split	**# Articles**	**# Sentences**
dev	281	997
devtest	281	1012
test	280	992
Domain	**# Articles**	**# Sentences**
WikiNews	309	993
WikiJunior	284	1006
WikiVoyage	249	1002
Sub-Topic	**# Articles**	**# Sentences**
Crime	155	313
Disasters	27	65
Entertainment	28	68
Geography	36	86
Health	27	67
Nature	17	45
Politics	171	341
Science	154	325
Sports	154	162
Travel	505	1529

Gambar 2.5.1 Statistik dari data dalam FLORES-101