BAB 2

LANDASAN TEORI

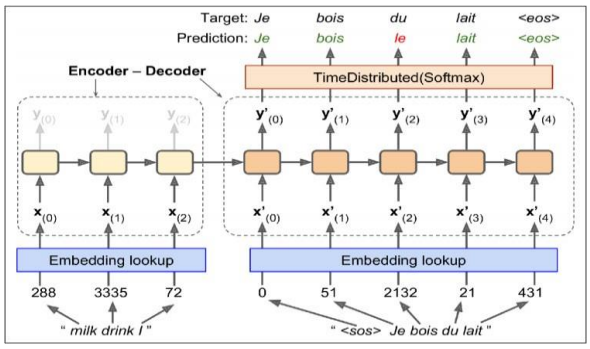
2.1 Difabel

Disabilitas adalah istilah umum untuk gangguan, keterbatasan aktivitas dan keterbatasan dalam partisipasi yang mengacu kepada aspek negatif dari interaksi individu (kondisi kesehatan) dan faktor kontekstual (lingkungan dan pribadi) (Vornholt *et al.*, 2018). ILO (2014) menyatakan bahwa penyandang disabilitas adalah orang-orang yang memiliki gangguan pada fisik, mental, intelektual, dan sensoris yang menghalangi keikutsertaan dalam bermasyarakat. Kebanyakan masyarakat memiliki pola pikir negatif tentang penyandang disabilitas. Menurut Fibrianto and Yuniar (2021) difabel (*diffable - differently ability)* artinya orang yang melakukan aktivitas dengan cara yang lain. Dengan penggunaan kata difabel, pola pikir masyarakat terhadap difabel dapat menjadi lebih baik.

2.2 Neural Machine Translation

Neural Machine Translation (NMT) merupakan model yang dibangun dengan tujuan melakukan translasi dari suatu bahasa ke bahasa yang lain. NMT terdiri dari *embedding layer*, *encoder*, dan *decoder*. *encoder* dan *decoder* merupakan RNN (*Recurrent Neural Network*) satu arah. Cara kerja dari model ini adalah membuat *word embedding* dari sebuah *input* dengan *embedding layer*. *Encoder* kemudian menghasilkan *word vector* dari *word embedding* yang kemudian dilanjutkan ke

decoder. Pada setiap langkah, *decoder* akan mengeluarkan skor untuk setiap kata yang dihasilkan, kemudian *softmax layer* mengubah skor-skor tersebut menjadi probabilitas. Kata dengan probabilitas terbesar akan menjadi hasil akhir. Kata ini juga akan digunakan sebagai *input* untuk *decoder* pada langkah selanjutnya (Géron, 2019).



Gambar 2.1 *Neural Machine Translation Model* (Géron, 2019)

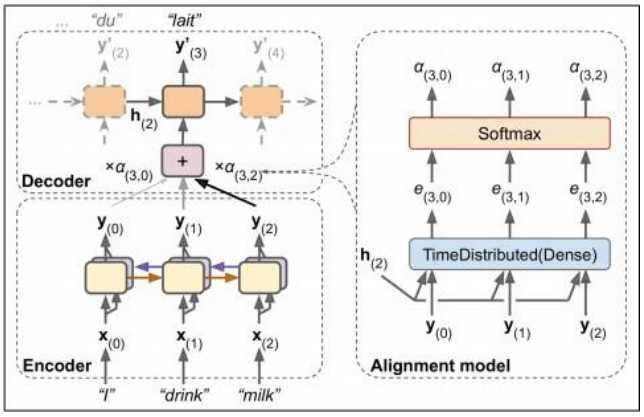
Pada setiap langkah, RNN hanya akan melihat dapat *input sebelumnya* sebelum menghasilkan hasil *output*. Dalam NLP (*Natural Language Processing*), konteks sebuah kata dalam kalimat sangat penting. *Bidirectional* RNN bekerja dengan cara satu RNN melihat *input* dari kiri ke kanan, dan RNN lainnya melihat *input* dari kanan ke kiri, kemudian menggabungkan keduanya.

2.3 Attention

Dengan *bidirectional RNN*, langkah yang diperlukan untuk menerjemahkan suatu kata sangat panjang. Diperkenalkan teknik untuk memperpendek langkah tersebut, yaitu *attention* yang memungkinkan *decoder* untuk fokus kepada satu kata (yang telah di-*encode*). Artinya, langkah yang diperlukan untuk menerjemahkan

sebuah kata menjadi lebih pendek, alhasil kekurangan *short-term memory* yang dimiliki RNN dapat diatasi (Géron, 2019).

Attention memungkinkan *decoder* untuk melihat seluruh *hidden state* dari *encoder*. Pada setiap langkah, *decoder* akan menghitung *weighted sum* dari setiap *output* yang dihasilkan *encoder*. Hasil dari *weighted sum* akan menentukan kata apa yang harus difokuskan oleh *decoder*. *Weight-weight* ini dihasilkan oleh *attention layer* (Bahdanau, Cho and Bengio, 2015; Géron, 2019). Pada Gambar 2.2, model *alignment* dimulai dengan *dense layer* yang menghitung kesesuaian antara *output* dari *encoder* dan *output* dari *decoder* pada langkah sebelumnya. Hasil skor tersebut akan melewati perhitungan *softmax*, untuk mendapatkan *weight* dari setiap *output* yang dihasilkan *encoder*.

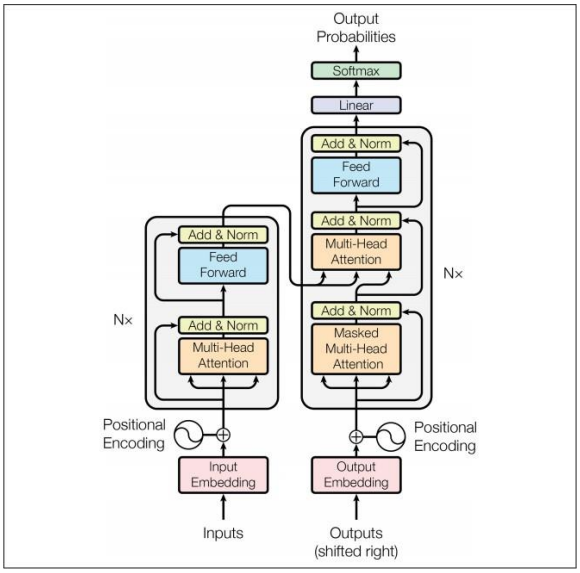


Gambar 2.2 NMT dengan *attention*
(Géron, 2019)

2.4 Transformer

Vaswani *et al.* (2017), merancang arsitektur yang disebut *transformer*. Arsitektur ini meningkatkan *state-of-the-art* NMT secara signifikan tanpa menggunakan *recurrent layer*. Pada arsitektur RNN, setiap kata pada kalimat akan diproses satu-persatu. Dengan menggunakan *transformer*, seluruh kata pada sebuah

kalimat akan diproses bersamaan secara paralel.



Gambar 2.3 Arsitektur *transformer*
(Vaswani *et al.*, 2017)

2.4.1 Encoder dan Decoder

Transformer memiliki dua bagian utama, yaitu *encoder* (sebelah kiri pada Gambar 2.3) dan *decoder* (sebelah kanan pada Gambar 2.3).

A. Encoder

Encoder memiliki dua *sub-layer*, yang pertama adalah *multi-head attention* dan yang kedua adalah *fully-connected feed-forward network*. *Multi-head attention layer* berguna untuk membantu *encoder* untuk fokus kepada suatu kata dan melihat konteks secara keseluruhan dari sebuah *input*. *Feed-forward network* akan menghasilkan vektor yang akan dilanjutkan ke *decoder* (Vaswani *et al.*, 2017).

B. Decoder

Arsitektur *decoder* sama seperti *encoder* dengan tambahan *sub-layer* ketiga, yaitu *masked multi-head attention*. *layer* ini akan melakukan menyembunyikan Sebagian vektor yang dihasilkan *encoder*. Hal ini berguna untuk mencegah *decoder* melihat kata selanjutnya. Dengan begitu, model tidak akan menyalin *input decoder*, tetapi mempelajari *output* yang dihasilkan *encoder*. *Input* pertama dari *decoder* adalah sebuah *token start*. Hal ini akan mencegah kekosongan dari pergeseran *output decoder*. *Token end* juga digunakan sebagai penanda akhir kalimat dan *decoder* tidak akan memproses *token end* tersebut (Vaswani *et al.*, 2017).

2.4.2 Self-Attention

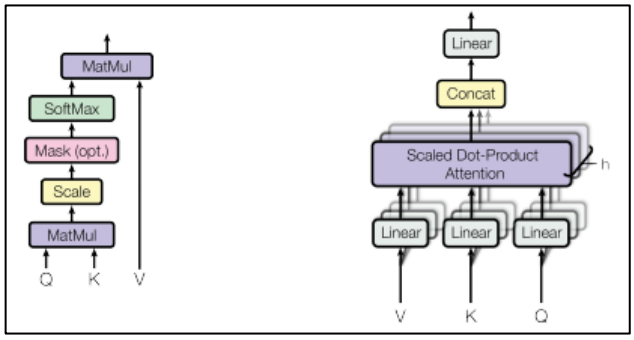
Self-attention pada *transformer* berguna sebagai panduan *transformer* untuk fokus kepada satu kata yang akan diproses. *Attention function* akan melakukan *mapping* dari tiga vektor yaitu *query, key dan value* (*key-value pair*) ke *output*. *Output* dari *self-attention* merupakan hasil dari *weighted sum* dari *values* (Vaswani *et al.*, 2017).

A. Scaled Dot-Product Attention

Input dari fungsi ini adalah *query* dan *key* dengan dimensi d_k dan *value* dengan dimensi d_v. Perhitungan dimulai dengan melakukan perkalian *dot product*

antara *query* dan *key*, kemudian membaginya dengan $\sqrt{d_k}$, kemudian melakukan perhitungan *softmax* untuk mendapatkan *weight*. *Weight* tersebut kemudian dikalikan dengan *value* (Vaswani *et al.*, 2017). Beberapa Vektor *query* akan dijadikan satu untuk menghasilkan matriks Q. *Key* dan *value* juga akan dijadikan satu untuk menghasilkan matriks K dan V. Berikut adalah notasi *attention function*.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$



Gambar 2.4 Alur *Scaled Dot-Product* dan alur *multi-head attention* (Vaswani *et al.*, 2017)

B. Multi-Head Attention

Multi-Head attention akan menginisialisasi matriks *query*, *key*, dan *value* sebanyak h kali. Pada penelitiannya, Vaswani menggunakan $h = 8$, sehingga hasil akhir adalah delapan matriks dengan nilai yang berbeda-beda. Karena *feed-forward network* hanya menerima 1 matriks *attention*, kedelapan matriks tersebut akan digabungkan dan dikalikan dengan matriks *weight*. Kemudian matriks tersebut akan dilanjutkan ke *feed-forward network* (Vaswani *et al.*, 2017).

C. Penggunaan Attention pada Transformer

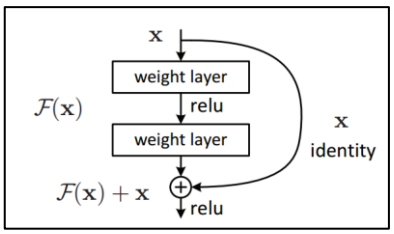
Transformer menggunakan *attention* dengan tiga cara berbeda. Penggunaan *attention* pada *transformer* adalah sebagai berikut (Vaswani *et al.*, 2017).

- Pada *encoder-decoder attention layer*, *query* dihasilkan oleh *decoder layer* sebelumnya, sementara *key* dan *value* dihasilkan oleh *output* dari *encoder*. Dengan begitu, setiap posisi di *decoder* dapat menempati *input sequence*.
- *Encoder* memiliki *self-attention layer* yang seluruh *key*, *value*, dan *query* dihasilkan dari *encoder* sebelumnya. Tiap posisi pada *encoder* dapat menempati seluruh posisi pada *layer* milik *encoder* sebelumnya.
- *Decoder* memiliki *self-attention layer* memungkinkan tiap posisi pada *decoder* untuk menempati seluruh posisi di *decoder* sampai dengan posisi tersebut. Untuk mencegah *attention* melakukan perhitungan dengan kata selanjutnya, isi dari matriks yang merupakan kata selanjutnya akan disembunyikan.

2.4.3 Residual Connection

Pada saat melakukan *training* pada *neural network* yang sangat besar, dapat terjadi degradasi *gradients*. Pada saat melakukan *back propagation* dari *layer* terakhir ke *layer* awal, *gradient descent* akan melakukan perkalian dengan matriks *weight*. Akibat banyaknya perkalian matriks, sebuah *gradient* dapat menghilang (nilainya menjadi 0) atau meledak (nilainya menjadi sangat besar). Dengan *neural network* yang semakin dalam, akurasi menurun dan meningkatkan *training error*

(He *et al.*, 2016). *Residual network* bekerja dengan cara menambahkan *residual mapping* dengan *mapping* yang diproses.



Gambar 2.5 *Residual Learning*
(He *et al.*, 2016)

Metode melangkahi data dari satu *layer* ke *layer* lain disebut dengan *shortcut connection* atau *skip connection*. Metode ini memungkinkan data mengalir lebih mudah karena tidak melewati *activation function* yang dapat melemahkan *gradient*.

2.4.4 Layer Normalization

Pada saat melakukan *training*, setiap *weight* akan di-*update* setiap saat mengakibatkan distribusi *layer* menjadi tidak normal. Semakin banyak *layer*, distribusi dari *weight* juga akan semakin tidak normal. Tujuan dari dilakukannya normalisasi *layer* adalah untuk menormalkan distribusi *weight* dari setiap layer dan mempercepat proses *training* (Ba, Kiros and Hinton, 2016).

2.4.5 Feed-Forward Network

Hasil vektor dari *attention layer* akan diteruskan ke *feed-forward neural network*. Setiap *encoder* dan *decoder* memiliki *feed-forward network* masing-masing

dengan arsitektur yang sama dan *parameter* yang berbeda untuk setiap *layer*. *Feed-forward network* tersebut terdiri dari dua transformasi linear dengan ReLU (*Rectified Linear Unit*) *activation function* di antara keduanya (Vaswani *et al.*, 2017). Notasi dari *feed-forward network* adalah sebagai berikut dengan x adalah *input*, W_1 dan W_2 adalah *weight*, b_1 dan b_2 adalah *bias*.

$$FFN(x) = \max(0, xW_1 + b_1)\, W_2 + b_2 \tag{2.2}$$

2.4.6 Embedding dan Softmax

Embedding dilakukan untuk mengubah *input* menjadi vektor dengan dimensi d_{model}. Vektor tersebut kemudian akan dilanjutkan ke *positional encoding*. Untuk mengubah *output* yang dihasilkan *decoder* menjadi *next-token probability*, digunakan *softmax function* dan *linear transformation*. *Linear transformation* dan kedua *embedding layer* memiliki matriks *weight* yang sama. *Weight tersebut* akan dikali dengan $\sqrt{d_{model}}$ pada *embedding layer* (Vaswani *et al.*, 2017).

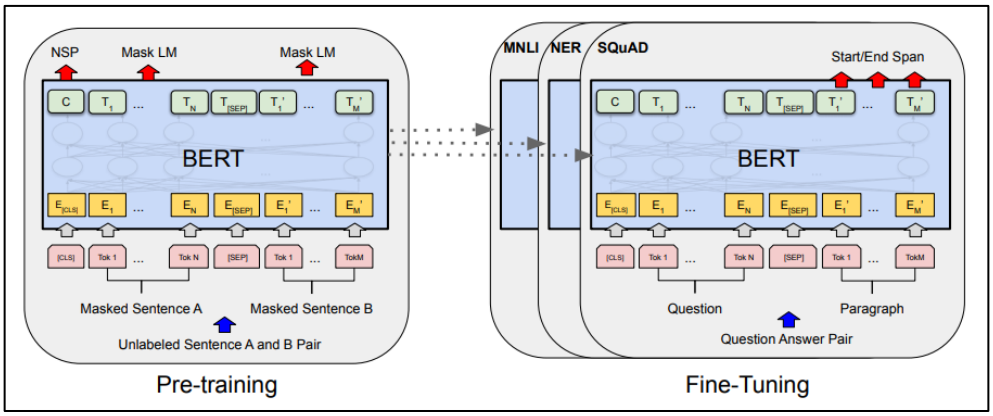
2.4.7 Positional Encoding

Model ini tidak menggunakan RNN (*Recurrent Neural Netword*) atau CNN (*Convolutional Neural Network*), sehingga untuk melakukan urutan kata pada kalimat, dimasukkan vektor *positional encoding* pada *input* yang diterima dari *embedding layer*. vektor *positional encoding* memiliki dimensi yang sama dengan vektor hasil *embedding* layer. Vektor tersebut kemudian digabungkan ke vektor hasil *embedding layer* (Vaswani *et al.*, 2017).

2.5 BERT

BERT (*Bidirectional Encoder Representation from Transformers*) merupakan *language* model yang di-*pretrain* menggunakan *dataset* yang sangat besar. Model ini dirancang menggunakan dan diperkenalkan pada tahun 2019 (Devlin *et al.*, 2019). Perbedaan model BERT dan model *directional a*dalah model *directional* menggunakan arah (kiri ke kanan, kanan ke kiri, atau gabungan keduanya). BERT menggunakan arsitektur *transformer* yang memungkinkan untuk melihat seluruh kata pada suatu *input*. Hal ini memungkinkan BERT untuk melihat keseluruhan konteks dari kalimat. *Transformer* memiliki dua bagian utama, yaitu *encoder* dan *decoder*. Tujuan BERT hanya membuat *language model*, sehingga BERT hanya menggunakan bagian *encoder*. BERT pertama kali di-*release* pada tahun 2018 dan memiliki dua ukuran. Yang pertama adalah $BERT_{BASE}$ dengan ukuran L=12, H=768, A=12, Total Parameter=110M. Model yang kedua yaitu $BERT_{LARGE}$ memiliki ukuran L=24, H=1024, A=16, Total Parameters=340M. dengan L adalah jumlah *layer*, H adalah *hidden size*, A adalah jumlah *self-attention head*.

Ada dua tugas dalam menggunakan BERT, yaitu *pre-training* dan *fine-tuning*. Dalam masa *pre-training*, model akan di-*train* dengan menggunakan data yang tidak berlabel. Dalam masa *fine-tuning*, BERT akan diinisialisasi dengan *parameter* hasil *pre-train*, kemudian *parameter-parameter* tersebut akan di-*fine-tune* menggunakan data berlabel sesuai dengan *downstream task masing-masing*. Setiap *downstream task* memiliki model yang telah di-*fine-tune* masing-masing, walaupun diinisialisasi dengan *parameter pre-train* yang sama.



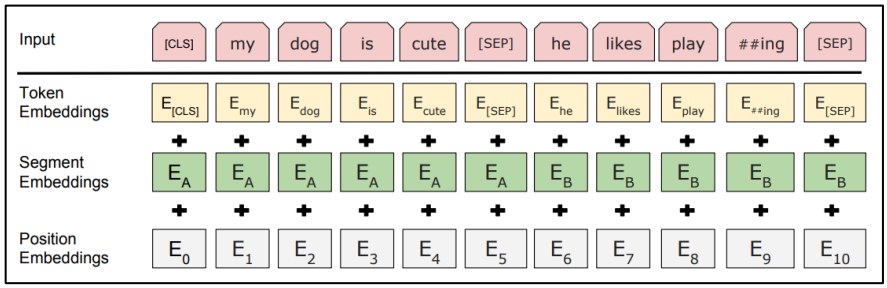
Gambar 2.6 *Pre-training* dan *fine-tuning* BERT
(Devlin *et al.*, 2019)

2.5.1 Pre-Training BERT

BERT menggunakan dua *unsupervised task*, yaitu *Masked Language Modeling* dan *Next Sentence Prediction* (Devlin *et al.*, 2019). Pada saat menerima *input*, BERT akan memilih sebanyak 15% kata secara *random* dari *input* dengan menggantinya dengan *token* [MASK]. BERT kemudian akan melakukan prediksi dari *token* tersebut berdasarkan konteks yang diberikan kata lain yang tidak di-*mask*. Dalam hal ini, *hidden vector* terakhir yang berkaitan dengan *token* [MASK] akan diberikan kepada *softmax function*. Tujuan dari dilakukannya MLM adalah membuat BERT menjadi mengerti konteks dua arah. Untuk melakukan training model yang mengerti hubungan antara kalimat, dilakukan *pre-training* terhadap *next sentence prediction* yang telah bagi menjadi dua. BERT kemudian melakukan prediksi apakah kedua kalimat tersebut berhubungan. Hal ini membuat BERT mengerti konteks dari kalimat-kalimat yang berbeda. Pada saat melakukan *pre-training*, *Masked Language Modeling* dan *Next Sentence Prediction* akan di-*train* secara bersamaan.

BERT tidak langsung menerima *input* berupa kalimat, tetapi kalimat tersebut akan melewati proses *embedding*. Proses *embedding* ini menggunakan tiga jenis *embedding*, yaitu *token embedding, segment embedding,* dan *position embedding*. *Token embedding* akan mengubah kata menjadi vektor *token* dengan menggunakan WordPiece *tokenization*. WordPiece dapat melakukan *tokenization* sampai kepada imbuhannya. *Tokenization* ini sangat berguna untuk membantu model dalam mengerti konteks kalimat (Wu *et al.*, 2016).

Token pertama dari setiap vektor adalah *token* [CLS] yang berarti *classification*. *Token* ini digunakan untuk melakukan klasifikasi teks. Jika nantinya BERT digunakan untuk melakukan hal lain, *token* ini akan diabaikan. Jika *input* berupa sepasang kalimat, digunakan *token* [SEP] untuk membedakan setiap kalimat. *Segment embedding* adalah nomor urut dari setiap kalimat yang di-*encode* ke vektor. *Positional embedding* adalah nomor urut dari setiap kata yang di-*encode* ke vektor.



Gambar 2.7 *Embedding* dari BERT terdiri dari tiga *embedding* (Devlin *et al.*, 2019)

Pada Gambar 2.7, *embedding* dinotasikan sebagai E. Vektor terakhir (*output vector*) akan mengubah *token* [CLS] menjadi C yang akan digunakan sebagai label atau kelas akhir, dan setiap *token* E, akan dihasilkan prediksi *token* T. Dengan begitu, jumlah *output token* yang dihasilkan sama dengan jumlah *input token*.

2.5.2 Fine-Tuning BERT

Tujuan dari *fine-tuning* BERT adalah menggunakan BERT yang sudah di-*pretrain* untuk menyelesaikan suatu masalah. BERT yang sudah di-*pretrain* dapat digunakan untuk menyelesaikan beberapa masalah seperti sistem penerjemah, *Question Answering System* (QAS), analisis sentimen atau klasifikasi teks, sistem peringkasan teks, dan *Named Entity Recognition* (NER) (Devlin *et al.*, 2019).

2.6 IndoBERT

Model BERT dapat digunakan dengan dua cara, yaitu *pre-training* dan *fine-tuning*. Untuk dapat melakukan *fine-tuning*, model tersebut harus sudah di-*pre-pretrain* dengan menggunakan *dataset* dengan bahasa yang sama. Model-model yang ada biasanya menggunakan bahasa yang telah memiliki korpus yang besar dan siap digunakan (Wilie *et al.*, 2020). Willie *et al*. telah merancang dan membangun BERT berbahasa Indonesia dengan nama IndoBERT.

IndoBERT dirancang berdasarkan arsitektur BERT, tetapi di-*pretrain* dengan menggunakan korpus Indo4B dengan dua fase. Fase pertama menggunakan *Maximum Sequence Length* sebesar 128 dan fase kedua menggunakan *Maximum Sequence Length* sebesar 512. Fase kedua memungkinkan IndoBERT mempelajari konteks lebih luas daripada fase pertama. *Dataset* Indo4B ini dikumpulkan dari berita *online*, artikel *online*, media sosial, dsb. Korpus ini memiliki 4 Miliar kata dan sekitar 250 juta kalimat. Dari hasil penelitian yang dilakukan Willie *et al*. (2020), model yang dibangun lebih unggul dalam beberapa *task*. Model IndoBERT

memiliki skor tertinggi untuk *task* klasifikasi teks dibandingkan dengan *task-task* lainnya.

2.7 ADAM

Adam adalah algoritma optimisasi untuk *neural network*. Algoritma ini dikembangkan oleh Diederik P. Kingma dan Jimmy Lei Ba (Kingma and Ba, 2015). Adam merupakan gabungan dari *Gradient Descent* dengan *RMSProp*. Adam menghitung *weighted moving average* dari gradien, kemudian mengkuadratkannya. Hasil dari penelitian yang dilakukan oleh Diederik P. Kingma dan Jimmy Lei Ba menunjukkan bahwa Adam dapat melakukan optimisasi lebih baik daripada *Gradient Descent* ataupun *Stochastic Gradient Descent*.

2.8 Back-Translation

Back-translation adalah sebuah metode augmentasi data teks. Tujuan dari data ini adalah memperbanyak dataset tanpa mengubah makna dari sebuah teks. Cara kerja metode *back-translation* adalah menggunakan *platform* penerjemah (seperti Google Translate) untuk menerjemahkan teks ke bahasa lain, kemudian teks hasil terjemahan akan diterjemahkan lagi ke bahasa awal (Edunov *et al.*, 2020). Contoh kata “selamat siang” dalam Bahasa Indonesia jika diterjemahkan ke Bahasa Jepang akan menjadi “こんにちは” (kon'nichiwa). Jika “こんにちは” diterjemahkan lagi ke Bahasa Indonesia, hasil terjemahannya adalah “Halo”. “Selamat Siang” dan “Halo” memiliki makna yang sama yaitu untuk menyapa.

Back-translation dapat mengaugmentasi data teks dengan menjaga konteks yang ada pada data awal (Yu *et al.*, 2018).

2.9 Evaluasi

Digunakan beberapa metrik untuk mengevaluasi model IndoBERT dalam melakukan klasifikasi berita difabel. Metrik-metrik tersebut adalah *Confusion matrix, accuracy, precision, recall,* dan F1 s*core*. *Confusion matrix* adalah sebuah teknik evaluasi *meachine learning* yang berisi informasi tentang hasil kelas dari prediksi dan yang sebenarnya. *Confusion matrix* memiliki dua dimensi, yang pertama berisi kelas yang sebenarnya dan yang kedua berisi kelas hasil prediksi dari model *machine learning*. Matriks ini berguna untuk menghitung *accuracy*, *precision, recall* dan F_1 score (Deng *et al.*, 2016).

		Predicted			
		A_1	... A_j ...		A_n
Actual	A_1	N_{11}	N_{1j}		N_{1n}
	⋮ A_i ⋮	N_{i1}	... N_{ij} ...		N_{in}
	A_n	N_{n1}	N_{nj}		N_{nn}

Gambar 2.8 C*onfusion matrix*
(Deng *et al.*, 2016)

- *Accuracy* adalah persentase prediksi yang benar.

$$Accuracy = \frac{\sum_{i=1}^{n} N_{ii}}{\sum_{i=1}^{n} \sum_{j=1}^{n} N_{ij}} \quad (2.3)$$