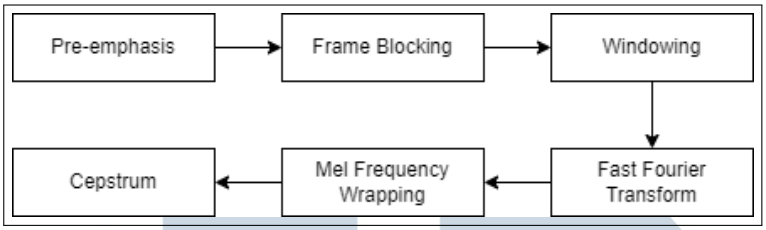
BAB 2
LANDASAN TEORI

2.1 GTZAN Dataset

Dataset yang akan digunakan untuk penelitian ini adalah GTZAN *dataset*. GTZAN *dataset* adalah *dataset* yang memiliki total 1000 lagu atau *file* audio. *Dataset* ini memiliki 10 genre, antara lain *blues*, klasik, *country*, *hip-hop*, *jazz*, *metal*, *pop*, *reggae*, dan *rock*. Masing-masing genre memiliki total 100 lagu dengan durasi 30 detik. *Dataset* ini pertama kali dibuat oleh George Tzanetakis dan Perry Cook pada tahun 2002 [9]. Awalnya *dataset* ini tersedia pada situs MARSYAS yang dibuat oleh Tzanetakis dan Cook [10], tetapi situs web tersebut sudah tidak tersedia sehingga *dataset* dapat diakses melalui Kaggle.

2.2 Mel-Frequency Cepstrum Coefficient (MFCC)

Mel-Frequency Cepstral Coefficient (MFCC) adalah salah satu metode ekstraksi fitur pada data audio yang dikenalkan oleh Davis dan Mermelstein pada tahun 1980. MFCC seringkali digunakan untuk *signal processing*, seperti *speaker recognition* dan *speech recognition* [11]. Ekstraksi suara yang dilakukan oleh MFCC berdasarkan pada perkiraan frekuensi suara yang dapat didengarkan oleh manusia. Sinyal yang digunakan pada MFCC adalah skala Mel, yaitu skala yang menggunakan filter linier pada frekuensi di bawah 1000 Hz dan jarak logaritmik di atas 1000 Hz [12]. Hasil dari MFCC berupa grafik gelombang spektrum atau spektogram berdasarkan frekuensi tersebut. MFCC memiliki kemampuan untuk mengekstraksi *feature* dari data suara tanpa menghilangkan atau mengurangi informasi penting pada data yang memiliki jumlah yang sedikit [13]. Hal ini menjadi alasan mengapa MFCC seringkali digunakan dalam mengekstraksi fitur. MFCC memiliki beberapa proses atau tahapan seperti yang diilustrasikan pada Gambar 2.1, yaitu *pre-emphasis*, *frame blocking*, *windowing*, *Fast Fourier Transform* (FFT), *Mel Frequency Wrapping* (MFW), *Discrete Cosinus Transform* (DCT), dan *cepstrum* [13].



Gambar 2.1. Tahapan *feature extraction* MFCC

2.2.1 Pre-emphasis

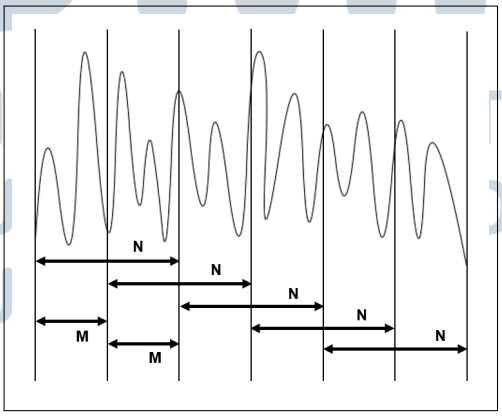
Pre-emphasis adalah tahap awal dari MFCC yang berfungsi sebagai filter untuk mengurangi *noise* dari data suara. Tujuan dari *pre-emphasis* adalah untuk menyeimbangkan sinyal suara terutama pada sinyal yang memiliki frekuensi tinggi. Selain itu, *pre-emphasis* menghasilkan spektrum sinyal frekuensi yang lebih halus.

$$y(n) = s(n) - \alpha s(n-1) \tag{2.1}$$

Persamaan 2.1 adalah persamaan yang digunakan untuk melakukan *pre-emphasis* [11]. $y(n)$ adalah nilai sinyal hasil *pre-emphasis*, $s(n)$ adalah nilai awal sinyal pada waktu ke-n, α adalah konstanta filter *pre-emphasis* yang dapat bernilai $0,9 - 1,0$.

2.2.2 Frame Blocking

Frame blocking adalah tahapan setelah *pre-emphasis* yang membagi sinyal menjadi beberapa *frame*. *Frame* memiliki besar N sampel dan *frame* tersebut digeser sebesar M sampel sehingga didapatkan $N = 2M$ seperti pada Gambar 2.2.



Gambar 2.2. *Frame blocking*

$$f_1(n) = y(Ml + n) \tag{2.2}$$

Persamaan 2.2 adalah persamaan jumlah *frame blocking* [11]. $f_1(n)$ adalah hasil *frame blocking*, n adalah 0, 1, ..., N-1 dengan N sebagai jumlah sampel, M adalah panjang *frame*, l adalah 0, 1, ..., L-1 dengan L sebagai seluruh sinyal, y adalah hasil *pre-emphasis*.

2.2.3 Windowing

Windowing adalah tahapan untuk memperbaiki *frame* suara yang sudah dihasilkan pada tahapan sebelumnya ketika terjadi diskontinuitas pada setiap ujung *frame*. Terdapat tiga jenis *windowing*, yaitu *Rectangular Window*, *Hamming Window*, dan *Hanning Window* [11].

$$X(n) = f_1(n)w(n) \tag{2.3}$$

Persamaan 2.3 adalah persamaan fungsi *windowing* secara general [11]. $X(n)$ adalah nilai hasil perhitungan *windowing*, f_1 adalah hasil dari tahapan sebelumnya (*frame blocking*), n adalah 0, 1, ..., N-1 dengan N sebagai jumlah sampel per *frame*, dan $w(n)$ sebagai fungsi *window*. Pada penelitian ini, tipe *windowing* yang digunakan adalah *Hanning Window* karena *default* yang disediakan pada saat melakukan MFCC menggunakan *library* Librosa adalah *Hanning Window*.

$$w(n) = 0,5\left(1 - cos\left(\frac{2\pi n}{M-1}\right)\right) \tag{2.4}$$

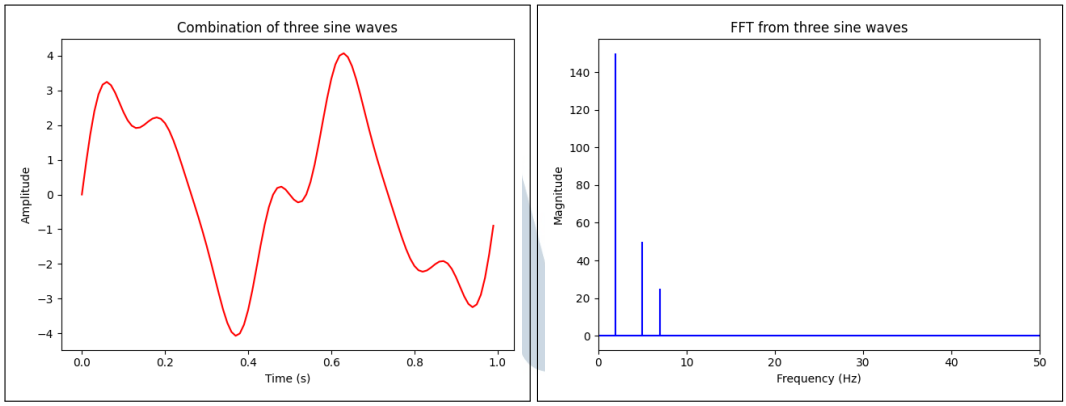
Persamaan 2.4 adalah persamaan *Hanning Window* [11]. $w(n)$ adalah hasil perhitungan *windowing* menggunakan *Hanning Window*, n adalah 0, 1, ..., M-1 dengan M sebagai panjang *frame*.

2.2.4 Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) merupakan algoritma yang menghitung *Discrete Fourier Transform* (DFT) dengan lebih cepat. Sebelum mendalami pengertian FFT, perlu diketahui terlebih dahulu mengenai *Fourier Transform* (FT). FT adalah konsep matematis yang digunakan untuk mengubah sinyal dari domain waktu (*time domain*). FT lebih sering digunakan untuk mengolah sinyal analog

[14], sedangkan DFT lebih digunakan untuk mengolah sinyal digital [15]. DFT menghitung spektrum sinyal tersebut dengan durasi yang terbatas [16]. Perbedaan dari FT dan DFT terletak pada *input* yang diterima. FT menerima *input* berupa sinyal kontinu, sedangkan DFT menerima *input* berupa sinyal diskret. Kelemahan dari DFT yaitu memiliki kompleksitas waktu $O(N^2)$ [17]. Hal ini membuat DFT lama dalam melakukan komputasi. Oleh karena itu, DFT kembali dikembangkan dan menghasilkan *Fast Fourier Transform* (FFT).

FFT pertama kali dikembangkan oleh James Cooley dan John Tukey pada tahun 1965. FFT adalah algoritma yang dapat menghitung DFT lebih cepat dengan kompleksitas waktu $O(Nlog_2N)$ tanpa mengurangi kompleksitas waktu dari DFT itu sendiri [16]. Contoh sederhana dari FFT dapat dilihat pada Gambar 2.3. Gambar 2.3a merupakan gelombang yang terdiri dari tiga frekuensi, yaitu 1 Hz, 4 Hz, dan 7 Hz dengan amplitudo masing-masing 3 m, 1 m, dan 0,5 m serta *sampling rate* 100 Hz. Gambar 2.3b menggambarkan hasil FFT dari gelombang sinyal pada Gambar 2.3a. Pada Gambar 2.3 dapat dilihat bahwa pada saat frekuensi berada pada 2 Hz, 5 Hz, dan 7 Hz, terjadi lonjakan magnitudo. Spektrum yang ditampilkan hanya sampai frekuensi 50 Hz karena teori frekuensi Nyquist, yaitu *output* tertinggi dari suatu sinyal yang dapat direkonstruksi kembali sampelnya [18].

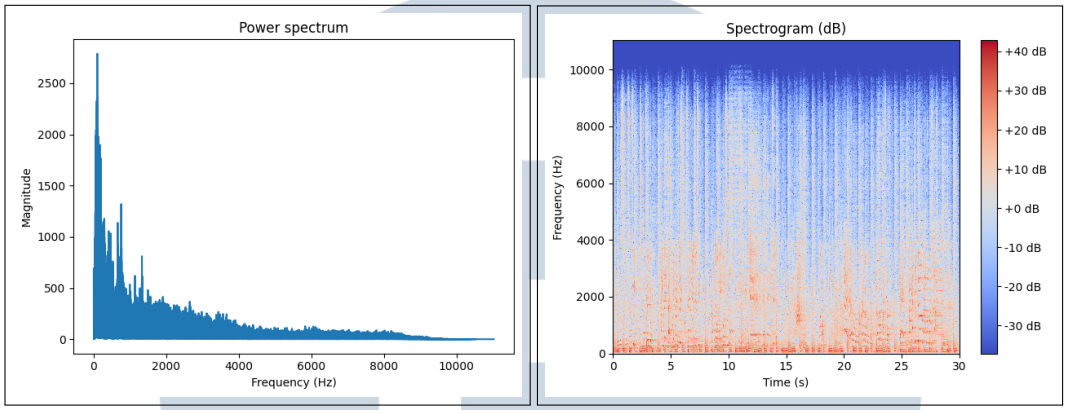


(a) Contoh gelombang sinyal sederhana (b) Spektrum sinyal setelah diubah ke FFT

Gambar 2.3. Contoh *Fast Fourier Transform* (FFT) pada sinyal sederhana

FFT tidak memberikan informasi mengenai waktu, sedangkan dalam memproses data suara, waktu sangat dibutuhkan karena setiap waktu dapat mengalami perubahan. Gambar 2.4a merupakan contoh spektrum dari sebuah cuplikan lagu dan dapat dilihat bahwa tidak ada informasi mengenai waktu. Oleh karena itu, digunakan *Short Time Fourier Transform* (STFT) untuk memberikan informasi tambahan berupa waktu sehingga hasilnya seperti pada Gambar 2.4b.

STFT menghasilkan spektogram yang berisi waktu, frekuensi, dan magnitudo dari data suara tersebut.

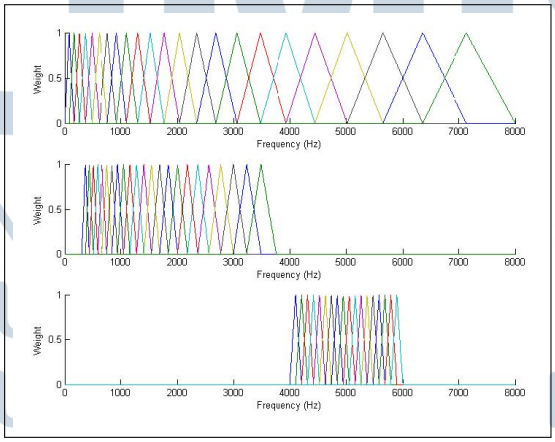


(a) Spektrum setelah melakukan FFT (b) Spektogram setelah dilakukan STFT

Gambar 2.4. Tahapan *Fast Fourier Transform* (FFT)

2.2.5 Mel Frequency Wrapping

Mel Frequency Wrapping (MFW) adalah tahapan yang menghasilkan spektrum Mel dengan menggunakan *filter bank*. *Filter bank* adalah filter yang digunakan untuk mendapatkan ukuran energi *frequency band* yang ada dalam sinyal suara [19]. Spektrum didapatkan pada tahap sebelumnya akan diaplikasikan *filter bank* dan menghasilkan spektrum Mel dengan menggunakan skala Mel atau *mel frequency scale*. Contoh spektrum Mel dapat dilihat pada Gambar 2.5.



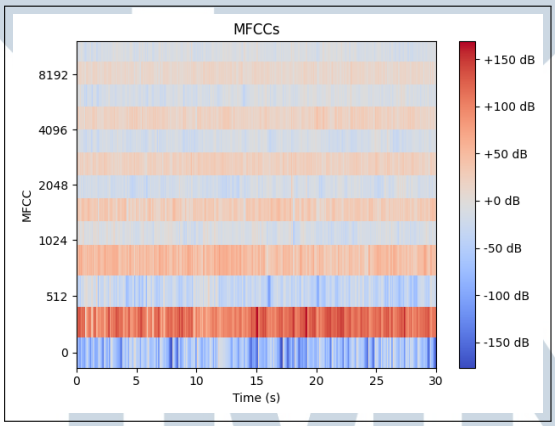
Gambar 2.5. Contoh spektrum Mel
Sumber: [11]

2.2.6 Cepstrum

Cepstrum adalah tahapan untuk mengambil informasi atau *feature* dari sinyal suara. Kata *cepstrum* diambil dari kata *spectrum* atau spektrum. Pada tahap ini, spektrum Mel yang dihasilkan oleh tahap sebelumnya akan diubah menjadi *cepstrum* dengan menggunakan *Discrete Cosine Transform* (DCT). Hasil akhir ini yang disebut dengan MFCC atau *Mel-Frequency Cepstrum Coefficient*.

$$C_m = \sum_{k=1}^{K} (log_{10} Y[k] cos\left[m(k - \frac{1}{2})\frac{\pi}{K}\right]) \tag{2.5}$$

Persamaan 2.5 adalah persamaan fungsi *Discrete Cosine Transform* (DCT) [11]. C_m adalah nilai MFCC, $Y[k]$ adalah hasil proses *filter bank* pada indeks k, m adalah 1, 2, ..., K dengan K sebagai jumlah koefisien yang diharapkan. Hasil MFCC dapat dilihat dalam bentuk spektogram seperti pada Gambar 2.6.

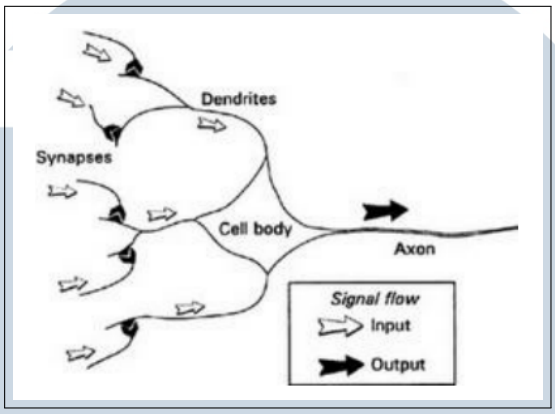


Gambar 2.6. Spektogram MFCC

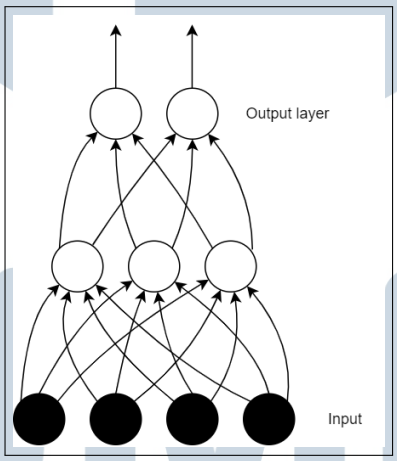
2.3 Deep Learning

Deep learning merupakan salah satu sub-bidang dari *machine learning* yang algoritmanya dibuat berdasarkan cara kerja otak manusia. Otak manusia memiliki neuron yang masing-masing berkomunikasi dengan memberikan sinyal elektrik ke dinding sel atau membran seperti pada Gambar 2.7 [20]. Hal ini yang menginspirasi model *deep learning* untuk menerapkan lapisan atau *layer* yang sering disebut sebagai *neural network* (NN) atau *Artificial Neural Network* (ANN). Tampilan *neural network* secara sederhana dapat dilihat pada Gambar 2.8. *Deep learning* memiliki kekuatan dan fleksibilitas yang lebih tinggi karena dapat memproses data

dengan jumlah yang besar dan tidak terstruktur. Data akan melalui lapisan-lapisan tersebut dan masing-masing lapisan akan mengekstraksi fitur secara progresif ke lapisan berikutnya [21].



Gambar 2.7. Cara neuron pada manusia berkomunikasi
Sumber: [20]



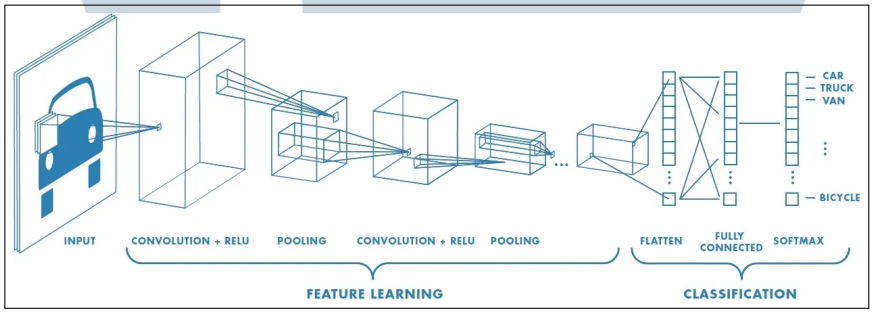
Gambar 2.8. Contoh sederhana dari *neural network*

Neural Network memiliki beberapa jenis algoritma, di antaranya yaitu *Convolutional Neural Network* (CNN) dan *Recurent Neural Network* (RNN). Algoritma yang akan digunakan pada penelitian ini adalah CRNN atau *Convolutional-Recurrent Neural Network*. Oleh karena itu, diperlukan pemahaman lebih lanjut mengenai CNN dan RNN agar dapat memahami algoritma tersebut.

2.3.1 Convolutional Neural Network (CNN)

Convolution Neural Network (CNN) adalah pengembangan dari salah satu Feed-Forward Neural Network (FFNN) yaitu Multi-Layer Perceptron (MLP).

Perbedaan dari MLP dan CNN sendiri terletak pada dimensinya, yaitu MLP hanya memiliki satu dimensi neuron, sedangkan CNN memiliki dua dimensi neuron. CNN terbagi menjadi dua bagian, yaitu bagian *feature learning* dan bagian *classification* [22]. *Feature learning* terdiri dari beberapa lapisan atau *layer*, antara lain *input layer*, *convolutional layer*, *activation layer*, dan *pooling layer*, sedangkan bagian *classification* terdiri dari *fully-connected layer* dan *output layer* [23]. Ilustrasi dari arsitektur CNN dapat dilihat pada Gambar 2.9. Hasil dari *feature learning* berupa angka-angka representasi gambar yang disebut *feature maps* yang kemudian akan dilanjutkan ke bagian *classification*. Setelah mendapatkan *input* dari bagian *feature learning*, dilakukan proses *flatten* yaitu penggabungan hasil *input* dan digabungkan melalui *fully-connected layer* yang akan menghasilkan skor akurasi dari masing-masing klasifikasi kelas.



Gambar 2.9. Arsitektur CNN
Sumber: [24]

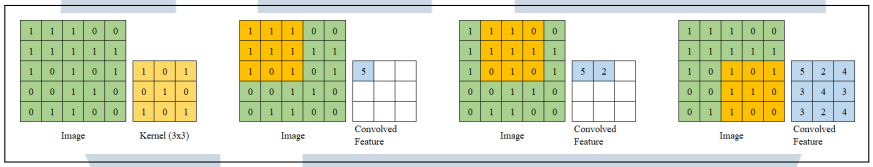
Berikut merupakan penjelasan mengenai beberapa lapisan atau *layer* utama yang digunakan pada algoritma CNN.

A Convolutional Layer

Convolution layer adalah *layer* pertama yang dilalui data pada arsitektur CNN. Pada *layer* ini, dilakukan operasi konvolusi yakni melakukan perkalian matriks pada *filter* secara berulang yang kemudian menghasilkan sebuah matriks baru. *Filter* memiliki panjang, tinggi, dan lebar sesuai dengan volume data yang dimasukkan. Pada saat mengolah data, operasi konvolusi ini melakukan perkalian antara kernel dengan citra masukkan pada semua *offset* yang memungkinkan seperti pada Gambar 2.10 [23]. Kotak hijau merupakan daerah yang akan dilakukan konvolusi. Kotak kuning merupakan *kernel* yang digunakan untuk melakukan perkalian. *Kernel* akan bergerak dari kiri ke kanan dan ketika sudah sampai ujung

akan bergerak turun ke bawah. Hasil dari *layer* ini disebut dengan *feature map*. Terdapat beberapa *hypermeter* yang digunakan pada *layer* ini yaitu *depth*, *stride*, dan *zero-padding* [22].

1. *Depth*: kedalaman atau jumlah *layer filter* pada operasi konvolusi.
2. *Stride*: jumlah pergeseran *filter* pada saat proses konvolusi.
3. *Zero-padding*: parameter untuk menentukan jumlah *pixel* yang ditambahkan di sekitar *input* gambar dan berisi nilai nol.



Gambar 2.10. Contoh operasi konvolusi

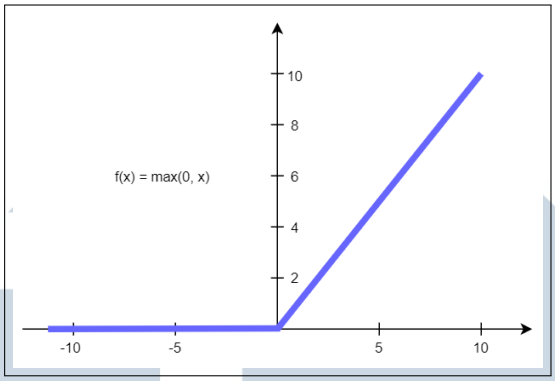
B Activation Layer

Activation layer adalah *layer* yang mengubah *feature map* menggunakan fungsi non-linier pada jarak tertentu berdasarkan pemilihan jenis fungsi aktivasi. Terdapat banyak jenis fungsi aktivasi, beberapa yang populer antara lain fungsi *sigmoid*, *tanh*, *Rectified Linear Unit* (ReLU), *Leaky* ReLU (LReLU), dan *Parametric* ReLU [22]. Namun, dari fungsi aktivasi yang populer, fungsi ReLU yang lebih sering dipakai pada penelitian. ReLU melakukan *thresholding* atau membatasi hasil dari 0 sampai *infinity* (tak terhingga) [23]. Fungsi aktivasi ReLU dapat dilihat pada Persamaan 2.6 dengan grafik seperti pada Gambar 2.6.

$$f(x) = max(0,x) \tag{2.6}$$

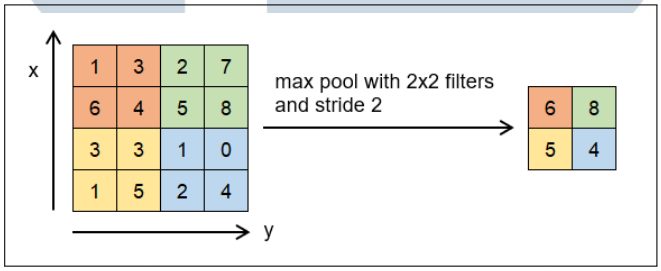
C Pooling Layer

Pooling layer adalah proses untuk mereduksi ukuran matriks pada *feature map* menggunakan operasi *pooling*. Operasi *pooling* akan melakukan pengambilan nilai secara rata-rata (*average pooling*) atau pengambilan nilai maksimal (*max pooling*). Contoh dari operasi *max pooling* dapat dilihat pada Gambar 2.12. Pada Gambar 2.12, operasi menggunakan *filter* berukuran 2×2 dengan jarak pergeseran



Gambar 2.11. Fungsi aktivasi ReLU

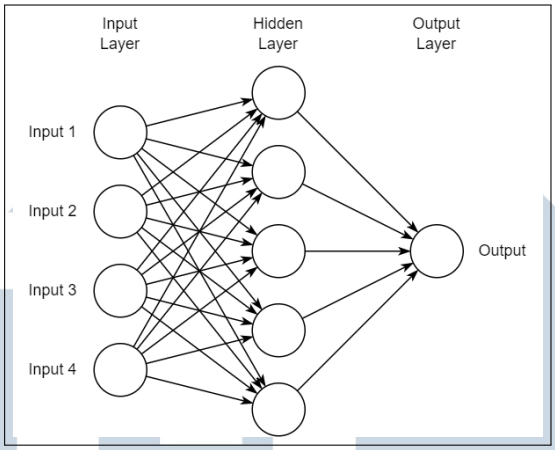
atau *stride* sebanyak 2 *kernel* yang menghasilkan matriks berukuran 2×2 yang sebelumnya berukuran 4×4. Operasi *pooling* digunakan untuk mengurangi jumlah parameter dan perhitungan pada jaringan serta untuk mengurangi kemungkinan *overfitting* [25].



Gambar 2.12. Contoh proses *max pooling*

D Fully-connected Layer

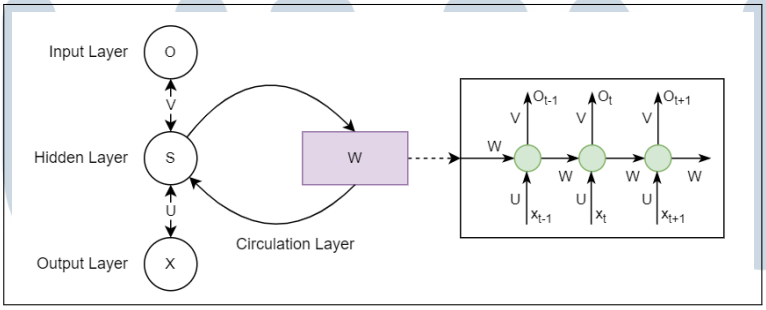
Fully-connected layer adalah *layer* yang menghubungkan semua neuron atau jaringan sebelumnya dengan neuron pada lapisan selanjutnya. Lapisan ini juga dikenal dengan sebutan *dense layer*. Neuron dari lapisan sebelumnya perlu diubah menjadi data satu dimensi agar dapat dihubungkan ke *fully-connected layer*. *Fully-connected layer* diimplementasikan pada akhir jaringan karena proses ini bersifat *irreversible* atau tidak reversibel dan menyebabkan data kehilangan informasinya [26]. Ilustrasi dari *fully-connected layer* dapat dilihat pada Gambar 2.13.



Gambar 2.13. *Fully-connected layer*

2.3.2 Recurrent Neural Network (RNN)

Recurrent Neural Network (RNN) adalah algoritma *neural network* yang dapat mengolah data sekuensial karena jaringan RNN memiliki memori internal di dalamnya yang berfungsi untuk menyimpan informasi mengenai *input* sebelumnya. Oleh karena itu, RNN dapat memecahkan masalah di bidang *Natural Language Processing* (NLP) seperti *speech recognition*, model bahasa, terjemahan mesin, dan analisis deret waktu [27]. RNN akan melakukan iterasi pada arsitekturnya yang akan selalu menyimpan data dari masa lalu, seperti pada Gambar 2.14.



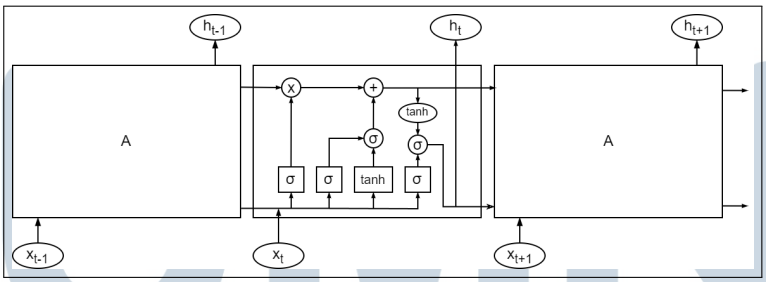
Gambar 2.14. Arsitektur RNN

Algoritma RNN terdiri dari tiga *layer*, yaitu *input layer*, *hidden layer*, dan *output layer*. *Input layer* (disimbolkan sebagai x_t) akan menerima *input* dan akan meneruskan *input* tersebut ke lapisan lainnya. *Hidden layer* (disimbolkan sebagai W) adalah lapisan tersembunyi yang akan menyimpan *input* untuk dijadikan *state* pada lapisan berikutnya dan akan memproses *input* dengan *hidden state* sebelumnya agar dapat menghasilkan sebuah *output*. *Output layer* (disimbolkan sebagai O_t) adalah lapisan hasil pemrosesan dari *input* dengan *hidden state* dari *input*

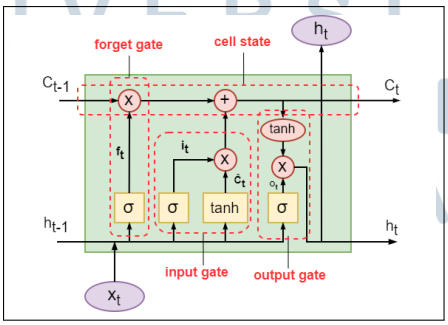
sebelumnya. Pada saat melakukan *training* atau pelatihan, RNN biasanya akan menggunakan algoritma *Backpropagation*. Gradien untuk setiap *output* tergantung pada kalkulasi dari *time step* saat ini dan sebelumnya karena parameter dibagikan secara merata pada setiap *time step* [28].

A Long Short Term Memory

Long-Short Term Memory (LSTM) merupakan algoritma pengembangan dari RNN. LSTM pertama kali dikenalkan oleh Hochreiter dan Schmidhuber pada tahun 1997. Karena RNN memiliki masalah pada *vanishing gradient* dan *explosion gradient*, LSTM merupakan solusi dari masalah tersebut. Masalah ini diselesaikan dengan memasukkan secara eksplisit ke dalam jaringan, sebuah sel sebagai unit. Sel ini bertugas sebagai *decision maker* dengan mempertimbangkan memori sebelumnya, *input* saat ini, dan output sebelumnya [29]. Oleh karena itu, pada arsitektur LSTM terdapat lapisan *input*, lapisan *output*, dan lapisan tersembunyi yang dapat dilihat pada Gambar 2.15. Skema dari dalam sebuah sel LSTM dapat dilihat pada Gambar 2.16. Dalam satu sel, terdapat tiga gerbang (*input*, *forget*, dan *output*), blok *input*, satu sel (*Constant Error Carousel*), fungsi aktivasi *output*, dan koneksi *peep-hole* [30].



Gambar 2.15. Arsitektur LSTM



Gambar 2.16. Diagram sel memori LSTM

Forget gate berfungsi untuk mengontrol jumlah informasi lama yang ingin disimpan pada sel memori. Gerbang ini mengambil *output* pada waktu $t-1$ dan *input* pada waktu t, menggabungkan kedua nilai tersebut dan kemudian diterapkan fungsi aktivasi *sigmoid*. *Sigmoid* akan memberikan *output* 0 atau 1. Ketika hasil f_t adalah 0, *state* sebelumnya akan dilupakan, sementara ketika f_t adalah 1, *state* sebelumnya tidak berubah.

$$f_t = \sigma(W_f \cdot h_{t-1} + W_f \cdot x_t + b_f) \tag{2.7}$$

Persamaan 2.7 adalah rumus untuk menghitung nilai *forget gate* [31]. f_t adalah nilai *forget gate*, σ adalah fungsi aktivasi *sigmoid*, W_f adalah bobot dari *forget gate*, h_{t-1} adalah *state* pada waktu $t-1$ (*timestep* sebelumnya), x_t adalah *input* pada waktu t, dan b_f adalah nilai bias pada *forget gate*.

Input gate pada LSTM berfungsi untuk mengontrol informasi yang masuk dan disimpan ke dalam sel serta mencegah sel untuk menyimpan data yang tidak perlu. Gerbang ini mengambil dan melewatkan *input* baru dan *output* sebelumnya ke lapisan *sigmoid*. Hasil dari gerbang ini adalah nilai 0 atau 1. Setelah itu, hasil akan dikalikan dengan *output* dari *candidate gate*. *Candidate gate* memuat kandidat nilai yang akan ditambahkan ke *cell state*.

$$i_t = \sigma(W_i \cdot h_{t-1} + W_i \cdot x_t + b_i) \tag{2.8}$$

$$\widetilde{C}_t = tanh(W_c \cdot h_{t-1} + W_c \cdot x_t + b_c) \tag{2.9}$$

Persamaan 2.8 adalah rumus yang digunakan untuk menghitung nilai *input gate* [31]. i_t adalah nilai *input gate*, σ adalah fungsi aktivasi *sigmoid*, W_i adalah bobot dari *input gate*, h_{t-1} adalah *state* sebelumnya, x_t adalah *input*, dan b_i adalah nilai bias pada *input gate*. Sementara itu, Persamaan 2.9 adalah rumus untuk menghitung nilai dari *candidate gate* [31]. $\widetilde{C}_t$ adalah nilai *candidate gate*, W_c adalah bobot dari *candidate gate*, dan b_c adalah bias pada *candidate gate*.

Setelah nilai dari *forget gate*, *input gate*, dan *candidate gate* sudah didapatkan, proses selanjutnya yang dilakukan adalah menghitung nilai *cell state*. *Cell state* adalah garis yang menghubungkan seluruh *output layer* pada sel LSTM. *Cell state* diubah dengan melakukan penjumlahan hasil perkalian antara *forget gate* dan *cell state* sebelumnya dengan hasil perkalian antara *input gate* dan *candidate gate*.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{2.10}$$

Persamaan 2.10 adalah rumus untuk mengubah *cell state* [31]. C_t adalah nilai *cell state*, f_t adalah nilai *forget gate*, C_{t-1} adalah *cell state* sebelumnya, i_t adalah nilai *input gate*, dan $\tilde{C}_t$ adalah nilai *candidate gate*.

Output gate berfungsi untuk memutuskan nilai yang digunakan untuk menghitung *hidden state* baru berdasarkan *input* dan *memory cell*. Gerbang ini juga digunakan untuk mengontrol berapa *state* yang masuk ke dalamnya.

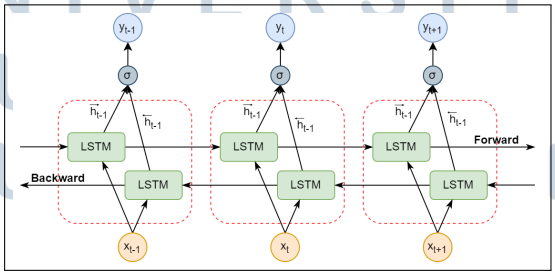
$$o_t = \sigma(W_o \cdot h_{t-1} + W_o \cdot x_t + b_o) \tag{2.11}$$

$$h_t = o_t * tanh(c_t) \tag{2.12}$$

Persamaan 2.10 adalah rumus untuk mendapatkan nilai *output gate* [32]. o_t adalah nilai *output gate*, W_o adalah bobot dari *output gate*, h_{t-1} adalah *state* sebelumnya, x_t adalah *input* pada waktu t, dan b_o adalah bias dari *output gate*. Persamaan 2.12 adalah persamaan untuk mendapatkan *hidden state* yang baru.

B Bidirectional Long Short Term Memory

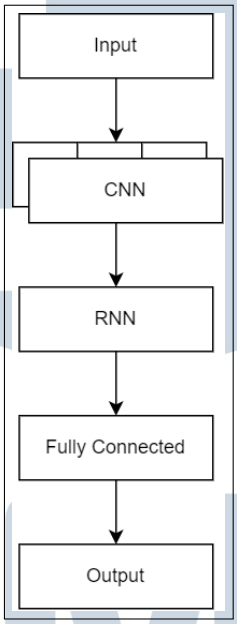
Bidirectional Long Short Term Memory (BiLSTM) adalah salah satu varian dari LSTM yang sering digunakan. BiLSTM memiliki dua jaringan LSTM yang berfungsi untuk mengolah urutan *input* secara *forward* dan *backward* [33]. Oleh karena itu, BiLSTM cocok digunakan untuk mengolah data yang membutuhkan konteks dari *input*. BiLSTM dapat memproses informasi masa lalu dan masa datang pada setiap *sequence*. Informasi masa lalu akan disimpan pada *backward layer*, sedangkan informasi masa datang disimpan pada *forward layer*. Arsitektur dari lapisan BiLSTM dapat dilihat pada Gambar 2.17.



Gambar 2.17. Arsitektur BiLSTM

2.3.3 Convolutional-Recurrent Neural Network (CRNN)

Convolutional-Recurrent Neural Network (CRNN) adalah algoritma yang menggabungkan CNN dan RNN. CNN akan melakukan ekstraksi fitur pada data dan RNN akan menringkas fitur yang sudah diekstrak pada tahap sebelumnya. *Layer* atau lapisan pada CRNN merupakan gabungan dari kedua algoritma tersebut karena seperti yang sudah dijelaskan sebelumnya, CRNN merupakan gabungan dari CNN dan RNN. CRNN memiliki *convolutional layer* dan *max pooling layer* seperti pada CNN, *recurrent layer* seperti pada RNN, *fully connected layer* dan *output layer* [34]. Ilustrasi arsitektur CRNN sederhana dapat dilihat pada Gambar 2.18.



Gambar 2.18. Arsitektur sederhana CRNN

Pada Gambar 2.18, lapisan CNN sudah termasuk dengan *convolutional layer* dan *pooling layer*. Pada lapisan RNN, terdapat beberapa tipe RNN yang dapat digunakan, seperti GRU (*Gated Recurrent Unit*) atau LSTM (*Long Short Term Memory*), tetapi tidak menutup kemungkinan tipe RNN yang lainnya dapat digunakan juga pada arsitektur ini. Setelah *input* sudah melalui lapisan CNN dan RNN, kemudian *input* akan masuk ke lapisan *fully-connected*. Fungsi *fully-connected layer* pada arsitektur ini sama seperti fungsi yang berada pada CNN. *Output layer* merupakan *lapisan* terakhir yang menghasilkan hasil pemrosesan dari lapisan-lapisan sebelumnya.