BAB 2
LANDASAN TEORI

2.1 Penelitian Terdahulu

Tabel 2.1. Penelitian Terdahulu Terkait Deteksi Video *Deepfake*

Penelitian 1	
Nama Peneliti & Tahun Terbit	S. Antad, V. Arthamwar, R. Deshmukh, A. Chame, and H. Chhangani (2024)
Judul	A Hybrid approach for Deepfake Detection using CNN-RNN
Jurnal/Prosiding	2024 OPJU International Technology Conference on Smart Computing for Innovation and Advancement in Industry 4.0 [9]
Metode	ResNeXt50 & LSTM
Dataset	Deepfake Detection Challenge Dataset (DFDC)
Hasil	Akurasi deteksi mencapai 92.1088% pada data video dengan 100 *frame*, namun menurun menjadi 84.6625% pada video dengan 10 *frame*
Penelitian 2	
Nama Peneliti & Tahun Terbit	Muhammad Indra Abidin, Ingrid Nurtanio, dan Andani Achmada (2022)
Judul	Deepfake detection in videos using Long Short-Term Memory and CNN ResNext
Jurnal/Prosiding	ILKOM Jurnal Ilmiah [13]
Metode	ResNext & LSTM
Dataset	Dataset Video *Deepfake* yang Tidak Disebutkan
Hasil	Akurasi meningkat signifikan dari 59.95% (10 *frames*) menjadi 97.55% (60 *frames*) seiring bertambahnya jumlah *frame* yang digunakan

Tabel 2.1. Penelitian Terdahulu Terkait Deteksi Video *Deepfake* (lanjutan)

Penelitian 3	
Nama Peneliti & Tahun Terbit	M. Brodarič, V. Štruc, and P. Peer (2024)
Judul	Cross-Dataset Deepfake Detection: Evaluating the Generalization Capabilities of Modern DeepFake Detectors
Jurnal/Prosiding	Proceedings of the 27th Computer Vision Winter Workshop (CVWW 2024) [10]
Metode	Xception
Dataset	FaceForensics++ dataset subsets (Deepfakes, Face2Face, FaceShifter, FaceSwap, & Neural Textures) along with two self-made augmented dataset InsightFace & DiffFace
Hasil	Penelitian ini menunjukkan bahwa suatu model yang dilatih pada *dataset* salah satu metode pembuatan video *deepfake* dapat memiliki penurunan dalam akurasi yang cukup signifikan apabila mendeteksi video *deepfake* yang dibuat dengan metode berbeda
Penelitian 4	
Nama Peneliti & Tahun Terbit	P. Saikia, D. Dholaria, P. Yadav, V. Patel, & M. Roy (2022)
Judul	A Hybrid CNN-LSTM model for Video Deepfake Detection by Leveraging Optical Flow Features
Jurnal/Prosiding	Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN) [14]
Metode	CNN, LSTM, Optical Flow Features
Dataset	Celeb-DF, Deepfake Detection Challenge Dataset (DFDC), & FaceForensics++
Hasil	Model mendapatkan akurasi 91.21% pada *dataset* FaceForensics++, 79.49% pada *dataset* Celeb-DF, dan 66.26% pada *dataset* DFDC

Tabel 2.1 menyajikan ringkasan beberapa penelitian terdahulu terkait deteksi video *deepfake* berbasis *deep learning*. Antad et al. mengembangkan pendekatan hibrida menggunakan ResNeXt50 dan LSTM pada *dataset* DFDC [9]. Pendekatan

tersebut menghasilkan akurasi tinggi sebesar 92.1% pada video dengan 100 *frame*, namun mengalami penurunan signifikan menjadi 84.7% pada video dengan hanya 10 *frame*. Hal ini menyoroti pentingnya jumlah *frame* dalam mendeteksi inkonsistensi temporal.

Penelitian serupa dilakukan oleh Abidin et al. dengan menggunakan arsitektur ResNeXt dan LSTM [13]. Pendekatan tersebut menunjukkan tren serupa: peningkatan jumlah *frame* dari 10 ke 60 menghasilkan lonjakan akurasi dari 59.95% menjadi 97.55%, menunjukkan bahwa pemanfaatan informasi temporal dapat memberikan dampak yang signifikan terhadap performa deteksi.

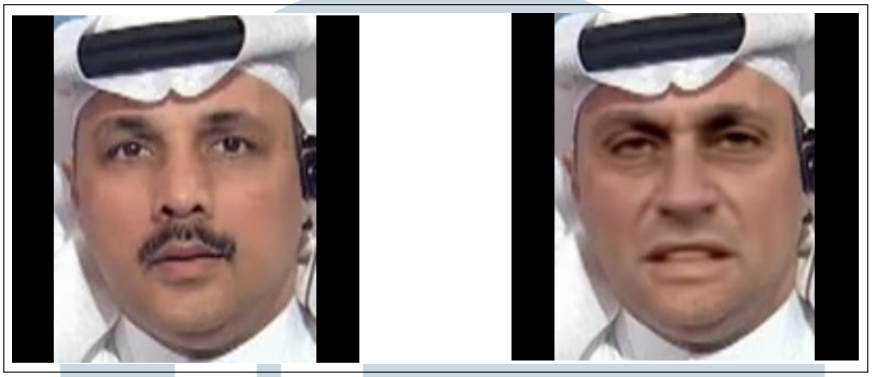
Sementara itu, Brodarič et al. menyoroti persoalan generalisasi lintas metode [10]. Dengan menggunakan arsitektur Xception pada *dataset* FaceForensics++, mereka menemukan bahwa model deteksi video *deepfake* cenderung mengalami penurunan performa ketika diuji pada metode pembuatan video *deepfake* yang berbeda dari data pelatihannya. Hal ini mempertegas pentingnya strategi pelatihan yang memperhatikan generalisasi.

Di sisi lain, Saikia et al. menggabungkan CNN dan LSTM dengan fitur *optical flow*, dan berhasil mencapai performa tinggi pada FaceForensics++ (91.21%) [14]. Namun, akurasi menurun pada *dataset* lain seperti Celeb-DF (79.49%) dan DFDC (66.26%), kembali menegaskan tantangan dalam hal generalisasi antar *dataset*. Secara keseluruhan, berbagai penelitian tersebut memperlihatkan tantangan utama dalam deteksi video *deepfake* sebagai berikut.

1. Sensitivitas model terhadap jumlah *frame* yang digunakan.
2. Kesulitan dalam generalisasi antar metode manipulasi video *deepfake*.

Maka penelitian ini akan mengeksplorasi pengaruh variasi metode pembuatan video *deepfake* terhadap performa deteksi model yang menggunakan arsitektur InceptionV3 dan LSTM. Berbeda dengan penelitian terdahulu yang umumnya melatih model secara terpisah untuk setiap jenis metode manipulasi, penelitian ini juga menambahkan satu model yang dilatih menggunakan gabungan berbagai jenis metode pembuatan video *deepfake*, namun dengan jumlah total video pelatihan yang sama. Perbandingan ini bertujuan untuk menganalisis pengaruh keragaman metode manipulasi dalam data pelatihan terhadap akurasi model, khususnya dalam konteks deteksi *cross-subset*.

2.2 *Deepfake*

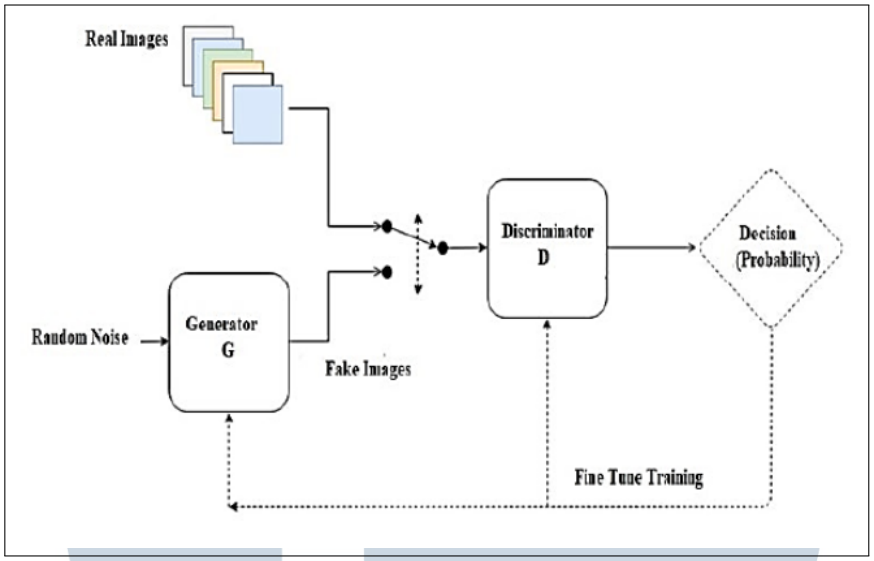


Gambar 2.1. Contoh Wajah Asli (Kiri) dan Wajah Hasil *Deepfake* (Kanan)
Sumber: [15]

Deepfake adalah rekayasa digital berbasis kecerdasan buatan yang menghasilkan suatu media yang telah dimanipulasi dalam bentuk gambar, video, atau rekaman video seperti pada gambar 2.1 [1]. Hasil manipulasi tersebut dapat menunjukkan seseorang seolah-olah melakukan atau mengatakan sesuatu yang sebenarnya tidak pernah terjadi. Terminologi *deepfake* berasal dari kata *deep learning* dan *fake* yang pertama kali diperkenalkan oleh seorang pengguna anonim Reddit pada akhir tahun 2017 ketika pengguna tersebut mengunggah sebuah video pornografi dengan muka orang lain yang sangat realistis [2].

2.2.1 *Generative Adversarial Network* (GAN)

Generative Adversarial Network atau GAN merupakan salah satu metode yang digunakan untuk menghasilkan suatu video *deepfake*. GAN terdiri dari dua jaringan saraf, yaitu *generator* G dan *discriminator* D seperti pada gambar 2.2 [16]. G akan menerima masukan berupa *random noise* N, yang kemudian diproses untuk menghasilkan media palsu yang menyerupai sampel dari *dataset* asli. Di sisi lain, *discriminator* D berfungsi untuk membedakan antara hasil yang dibuat oleh G dengan sampel asli. Dalam prosesnya, D akan mengevaluasi setiap sampel dan memperkirakan probabilitas sampel tersebut untuk berasal dari data asli atau dihasilkan oleh G.



Gambar 2.2. Struktur *Generative Adverserial Network* (GAN)
Sumber: [16]

D akan menghasilkan nilai probabilitas 1 ketika tingkat keyakinan bahwa media tersebut asli dan 0 pada media palsu [16]. D bertujuan untuk memaksimalkan jumlah jenis media yang diklasifikasikan dengan akurat, sedangkan G berusaha untuk membuat klasifikasi D menjadi kurang akurat. Maka, dapat dikatakan bahwa G dan D bermain sebuah permainan *minimax*, di mana D dilatih untuk memaksimalkan probabilitas untuk membedakan media asli dan palsu dengan akurat, sedangkan G berusaha untuk menipu D agar mendeteksi media palsu sebagai sebuah media asli.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{n\sim p_n(n)}[\log(1-D(G(z)))] \quad (2.1)$$

Persamaan matematis permainan *minimax* tersebut dapat diekspresikan pada persamaan 2.1 [16]. $p_{\text{data}}(x)$ merupakan distribusi media asli dan $p_n(n)$ adalah distribusi *noise*. Ketika pelatihan selesai dilakukan, G akan mampu untuk menghasilkan media palsu yang terasa natural dan realistis dengan sinyal noise n, sedangkan kemampuan D untuk membedakan media asli dan palsu juga akan meningkat.

2.2.2 *Autoencoder*

Autoencoder adalah sebuah jenis jaringan saraf tiruan yang berfungsi untuk mereproduksi suatu masukan secara *unsupervised* [17]. Tujuan dari *autoencoder* adalah melatih fungsi A : $\mathbb{R}^n \rightarrow \mathbb{R}^p$ *encoder* dan B : $\mathbb{R}^p \rightarrow \mathbb{R}^n$ yang memenuhi persamaan 2.2,

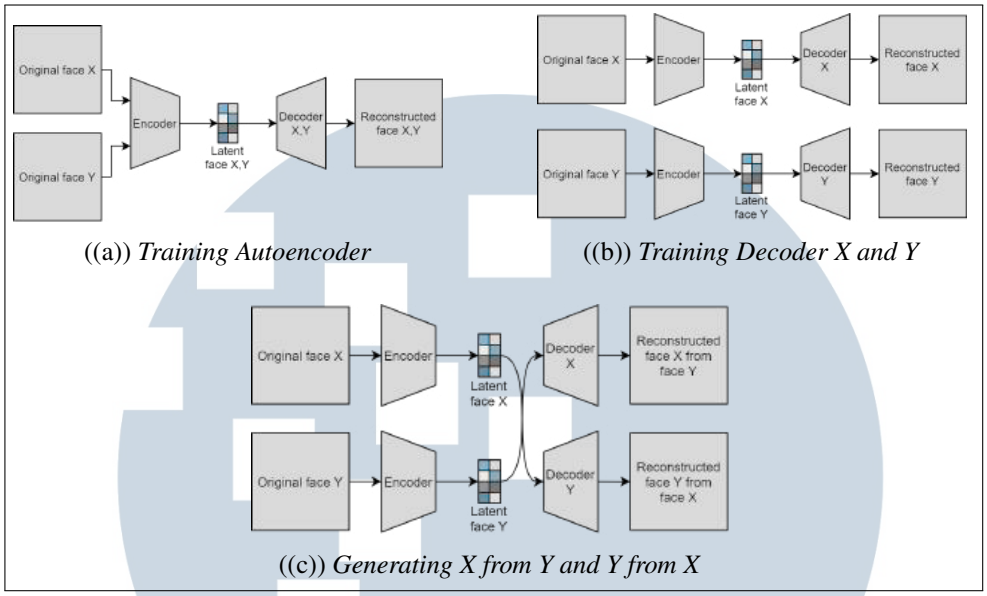
$$\arg\min_{A,B} \mathbb{E}[\Delta(x, (B \circ A)(x))] \tag{2.2}$$

di mana E merupakan ekspektasi dari distribusi data *input* x, Δ adalah fungsi *loss* rekonstruksi yang mengukur jarak antara *input* asli dan hasil rekonstruksi dari *decoder*, serta $(B \circ A)(x)$ merupakan komposisi fungsi yang merepresentasikan proses *encoding* dan *decoding* dari *input* x.

Biasanya, *autoencoder* terdiri dari berbagai *hidden layer* di antara *input* dan *output* layer dengan jumlah *neuron* yang lebih sedikit dibandingkan jumlah piksel pada *input image* [17]. Struktur ini menciptakan suatu *bottleneck* yang memaksa jaringan untuk mengompresi informasi dari *input* ke dalam bentuk representasi berdimensi rendah atau yang lebih dikenal dengan *latent representation* atau *feature map*. Dalam konteks *deepfake*, representasi ini sering disebut sebagai *latent face*.

Kompresi ini membuat representasi tersebut hanya mempertahankan informasi yang paling relevan untuk proses rekonstruksi dan mengabaikan data yang tidak diperlukan. Dengan demikian, *autoencoder* tidak hanya mereplikasi piksel *input* secara langsung, namun juga belajar untuk mengekstraksi fitur-fitur penting dari wajah yang kemudian dapat digunakan untuk menghasilkan *media* baru dengan ekspresi atau identitas yang berbeda.

Membuat *deepfake* dengan *autoencoder* terdiri dari tiga tahapan utama seperti yang ditunjukkan pada gambar 2.3 [17]. Misalkan terdapat dua himpunan citra wajah, yaitu X yang merepresentasikan wajah individu x, dan Y yang merepresentasikan wajah individu y. Pada tahap pertama (gambar 2.3(a)) , sebuah *encoder* yang bertugas untuk menghasilkan *feature map* dari kedua himpunan citra wajah tersebut. Di saat bersamaan, juga terdapat *decoder* (*decoder* XY) umum yang dilatih untuk merekonstruksi kembali citra wajah asli dari *feature map* tersebut sebagai tolok ukur akurasi dari *encoder*.



Gambar 2.3. Menghasilkan *Deepfake* Menggunakan *Autoencoder*

Sumber: [17]

Pada tahap kedua (gambar 2.3(b)), terdapat dua *decoder* terpisah, yaitu *decoder X* dan *decoder Y*, yang dilatih secara mandiri untuk menghasilkan citra wajah asli dari *feature map* yang telah dihasilkan oleh *encoder* yang telah dilatih sebelumnya [17]. Tujuan dari tahap ini adalah memungkinkan masing-masing *decoder* untuk mengenali dan mengembalikan karakteristik wajah individu spesifik berdasarkan *latent representation* yang sama.

Lalu pada tahap terakhir (gambar 2.3(c)), proses pembuatan *deepfake* dengan menukar *decoder* yang telah dilatih sebelumnya [17]. Sebagai contoh, citra wajah dari individu x diubah menjadi suatu *feature map* melalui *encoder*. Kemudian, decoder *Y* digunakan untuk merekonstruksi wajah tersebut dalam bentuk wajah individu y. Hasilnya adalah citra wajah yang mempertahankan ekspresi, pose, dan pencahayaan dari individu *x*, namun memiliki karakteristik wajah individu *y*. Proses sebaliknya juga dapat dilakukan untuk menghasilkan wajah individu *x* dari citra individu *y*.

UNIVERSITAS MULTIMEDIA NUSANTARA

2.2.3 FaceForensics++

Tabel 2.2. Deskripsi *Dataset* FaceForensics++

No	Nama	Asli	Palsu	Metode Pembuatan	Sumber
1	FaceForensics++	1000	1000	Deepfakes, Face2Face, FaceSwap, NeuralTextures, FaceShifter	[15], [18]

FaceForensics++ merupakan salah satu *dataset* paling populer yang terdiri dari 1000 video asli dan 5000 video palsu yang dibuat menggunakan lima metode manipulasi berbeda sebagai berikut seperti pada gambar 2.4.

Gambar 2.4. Contoh Teknik Pembuatan *Deepfake* pada *Dataset* FaceForensics++

1. **Deepfakes:** Metode ini menggunakan dua *autoencoder* dengan sebuah *encoder* bersama untuk merekonstruksi wajah target menggunakan fitur wajah dari video sumber. Wajah hasil rekonstruksi *autoencoder* kemudian digabungkan ke video *target* dengan menggunakan teknik *blending Poisson Image Editing*.
2. **Face2Face:** Metode ini hanya memindahkan ekspresi dari sebuah sumber video ke video *target* dan tetap mempertahankan identitas orang yang berada pada video *target*. Proses ini menggunakan pelacakan ekspresi wajah dan parameter pencahayaan.

3. **FaceSwap:** Metode ini melibatkan pelacakan *landmark* wajah dan pemetaan tekstur 3D dari wajah dari video sumber terhadap video *target*. Metode ini secara grafis mengganti seluruh area wajah melalui proyeksi 3D dan koreksi warna.

4. **NeuralTextures:** Metode ini menggabungkan *neural texture* dan *rendering network* berbasis GAN untuk mensintesis ekspresi wajah secara realistis dari video sumber terhadap video *target*. *Tracking module* yang digunakan untuk metode ini sama dengan Face2Face. Manipulasi difokuskan pada area mulut dengan tetap mempertahankan gerakan mata asli.

5. **FaceShifter:** Metode ini menggunakan arsitektur dua tahap untuk menghasilkan hasil yang realistis dan tetap mempertahankan atribut wajah target. Tahap pertama *Adaptive Embedding Integration Network* (AEI-Net) secara adaptif menggabungkan identitas wajah dari gambar sumber dan atribut wajah dari gambar target seperti pose, pencahayaan, dan ekspresi. Tahap kedua *Heuristic Error Acknowledging Refinement Network* (HEAR-Net) memperbaiki hasil dengan menangani area wajah yang mengalami *occlusion* secara *self-supervised*.

FaceForensics++ juga memiliki beberapa tingkat kualitas video sebagai berikut [15].

1. **RAW**
   - Video tidak terkompresi.
   - Kualitas video terbaik dengan artifak minimal.
   - Ukuran video *dataset* lengkap sekitar 500 GB.
   - Paling cocok untuk penelitian yang membutuhkan informasi visual tanpa modifikasi.
2. **C23 (*High Quality*)**
   - Video dikompresi dengan *codec* H.264 dengan *constant rate quantization parameter* 23.
   - Kualitas video moderat dengan kompresi yang masih mempertahankan detail visual yang baik.
   - Ukuran video *dataset* lengkap sekitar 10 GB.

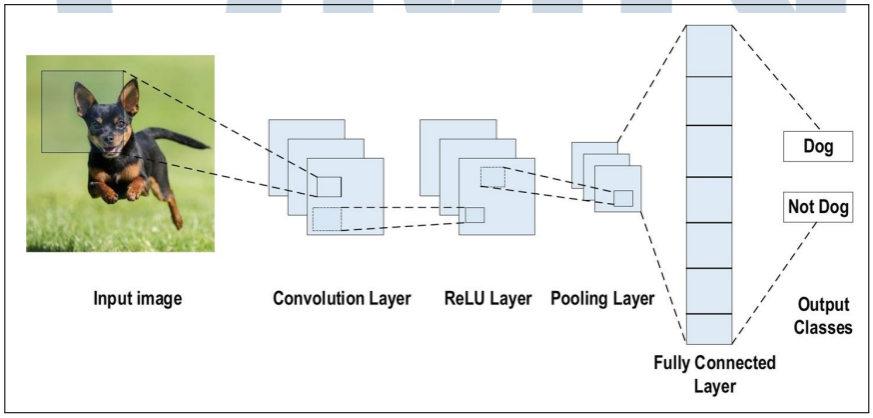
- Merepresentasikan kualitas video yang umum ditemukan pada platform video berkualitas tinggi.

3. **C40 (*Low Quality*)**
    - Video dikompresi dengan *codec* H.264 dengan *constant rate quantization parameter* 40.
    - Kompresi berat dengan artifak kompresi yang terlihat jelas.
    - Ukuran video *dataset* lengkap sekitar 2 GB.
    - Merepresentasikan video kualitas rendah yang sering ditemukan di media sosial setelah mengalami beberapa kali kompresi.

2.3 *Convolutional Neural Network* (CNN)

Convolutional Neural Network (CNN) adalah sebuah jenis model *deep learning* yang terinspirasi dari cara kerja neuron pada otak manusia dan hewan [19]. CNN sangat cocok untuk data dengan pola berbentuk *grid*, seperti gambar, serta telah banyak diterapkan dalam berbagai bidang seperti *computer vision*, *speech processing*, dan *face recognition* [19, 20].

Dibandingkan dengan model terdahulu, CNN memiliki keunggulan dalam kemampuannya untuk mengidentifikasi fitur-fitur penting tanpa supervisi langsung dari manusia [20]. Selain itu, CNN memiliki *weight sharing feature*, di mana parameter yang sama digunakan pada berbagai posisi input. Hal ini mengurangi jumlah parameter dan meningkatkan generalisasi serta mengurangi *overfitting*.



Gambar 2.5. Contoh Arsitektur CNN Untuk Klasifikasi Gambar

Sumber: [20]

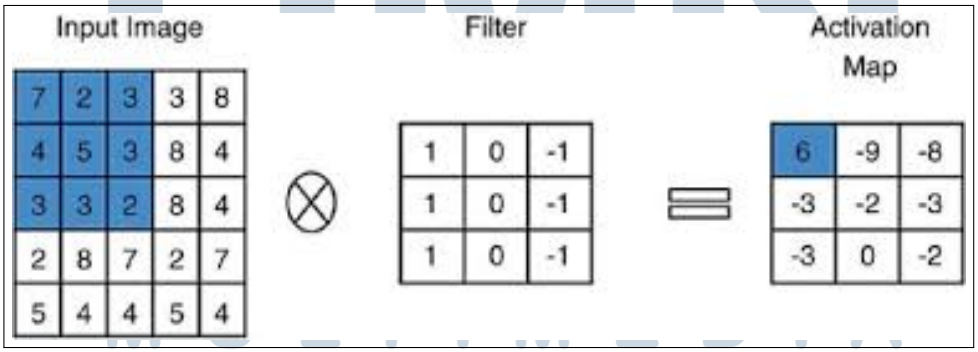
Berdasarkan cara kerjanya, CNN terdiri dari tiga lapisan utama, yaitu *convolutional layer*, *pooling layer*, dan *fully connected layer* seperti pada gambar 2.5. Lapisan-lapisan tersebut bekerja memiliki fungsi yang berbeda-beda untuk mengekstraksi fitur dan mengklasifikasikan objek dengan rincian sebagai berikut.

2.3.1 *Convolutional Layer*

Convolutional Layer adalah lapisan paling penting dalam CNN yang berfungsi untuk melakukan ekstraksi fitur [19]. Lapisan ini terdiri dari sekumpulan *convolutional filters* atau *kernels* yang bergerak melintasi data *input* yang direpresentasikan sebagai *tensor*, yaitu sebuah struktur data multidimensi yang dapat berupa vektor, matriks, atau *array* angka multidimensi.

Kernel merupakan kumpulan dari nilai beban yang akan digunakan untuk melakukan operasi konvolusi pada data *input*. Setiap *kernel* memiliki ukuran tetap yang lebih kecil dibandingkan dengan dimensi spasial (lebar dan tinggi) *input*, tetapi *channel* atau *depth* dapat sama atau lebih kecil dari *input*, tergantung pada konfigurasi jaringan [20].

Dalam satu *convolutional layer*, biasa digunakan beberapa *kernel* secara bersamaan. Masing-masing *kernel* bertanggung jawab untuk mengekstraksi fitur yang berbeda. Hal ini memungkinkan CNN untuk menangkap beragam pola dari suatu *input*, seperti tepi horizontal, sudut, atau tekstur yang kompleks. Hasil dari setiap *kernel* akan menghasilkan *feature map* tersendiri, yang kemudian digabungkan dan diproses lebih lanjut dalam jaringan [21].



Gambar 2.6. Operasi Konvolusi
Sumber: [22]

Operasi konvolusi bertujuan untuk mengekstraksi fitur penting dari *input* dengan melakukan *dot product* elemen-elemen *kernel* dengan nilai piksel yang

sesuai pada bagian gambar yang dilaluinya pada *image-based* CNN seperti pada gambar 2.6. Hasil operasi tersebut kemudian dijumlahkan untuk menghasilkan suatu nilai skalar yang akan mewakili suatu pola dalam gambar dan ditempatkan dalam sebuah *feature map*.

Selama proses konvolusi, *kernel* akan bergerak melintasi seluruh area *input* sesuai dengan parameter *stride* yang merupakan jarak perpindahan *kernel* dalam setiap langkahnya. Jika parameter *stride* bernilai 1, maka *kernel* akan berpindah senilai satu piksel setiap kali. Semakin besar nilai parameter *stride*, semakin kecil *feature map* yang akan dihasilkan karena lebih banyak informasi yang dilewati.

Selain itu, sering juga digunakan *padding*, yaitu penambahan nilai nol pada tepi *input* untuk mempertahankan ukuran *output* agar tetap sama dengan ukuran awal atau memastikan fitur tepi tetap diperhitungkan dalam proses konvolusi.

2.3.2 *Pooling Layer*

Pooling layer berfungsi untuk mengurangi dimensi dari *feature map* yang telah dihasilkan oleh lapisan konvolusi. Hal ini dilakukan untuk mengurangi parameter yang perlu dipelajari, invariansi terhadap pergeseran dan distorsi kecil pada data, serta mempertahankan informasi penting sambil menghilangkan detail yang kurang signifikan. Hal ini mempertahankan informasi yang dominan dalam setiap langkah pada *pooling stage*.

Pooling dilakukan dengan menggunakan *kernel* berukuran tetap yang bergerak melintasi *feature map* dengan langkah tertentu untuk melakukan operasi *downsampling*. Berbeda dengan lapisan konvolusi, *pooling layer* tidak memiliki parameter yang dapat dipelajari, tetapi bergantung pada *hyperparameter* seperti ukuran *kernel*, *stride*, dan *padding* yang ditentukan sebelum operasi *pooling* dilakukan.

Terdapat berbagai jenis *pooling* dalam CNN seperti teknik *max pooling* yang mengambil nilai maksimum dari sebuah area kecil atau *patch* dan membuang nilai lainnya. Terdapat juga teknik *average pooling* di mana nilai rata-rata dari satu *patch* dihitung. Terdapat juga teknik serupa bernama *global average pooling* yang melakukan hal serupa, namun diterapkan pada keseluruhan *feature map*. *Global average pooling* biasa dilakukan satu tahap sebelum mencapai *fully connected layers*.

Terkadang, performa model CNN dapat menurun karena lapisan ini [20]. Hal ini dikarenakan *pooling layer* menentukan apakah suatu fitur penting ada

atau tidak dalam suatu gambar atau *input* lainnya, namun dapat kehilangan detail mengenai informasi spasial fitur tersebut. Akibatnya, model CNN berpotensi untuk kehilangan informasi yang relevan, terutama pada tugas di mana lokasi dari fitur berperan signifikan.

2.3.3 *Fully Connected Layer*

Fully connected layer adalah lapisan yang biasanya terletak pada akhir arsitektur CNN. Lapisan ini berfungsi sebagai jembatan antara fitur yang telah diekstraksi model dengan hasil prediksi akhir. Sebelum masuk ke dalam lapisan ini, *output* dari lapisan konvolusi terakhir atau *pooling layer* yang berbentuk dua atau tiga dimensi diratakan terlebih dahulu menjadi *array* satu dimensi atau vektor. Setelah proses ini, *array* atau vektor diteruskan ke satu atau lebih *fully connected layers* atau *dense layers*, di mana setiap *input* dikoneksikan pada setiap *output* melalui bobot yang dapat dipelajari.

Dalam tugas klasifikasi, *fully connected layer* berperan dalam memetakan fitur-fitur yang telah diekstraksi dan diproses oleh CNN untuk menjadi *output* akhir, misalnya probabilitas dari masing-masing kelas dalam tugas klasifikasi. Lapisan *fully connected* biasanya memiliki jumlah *output node* yang sama dengan jumlah kelas yang ingin diprediksi. Selain itu, setiap *fully connected layer* biasanya diikuti suatu fungsi aktivasi non-linear, seperti RelU untuk menambahkan non-linearitas ke dalam jaringan, atau Softmax pada lapisan akhir untuk mengubah output menghasilkan probabilitas klasifikasi.

2.3.4 *Fungsi Aktivasi* (Non-Linier)

Fungsi aktivasi yang bersifat non-linier berfungsi untuk memetakan *input* terhadap *output* dalam suatu *neural network* [20]. Fungsi ini akan menentukan apakah suatu neuron akan teraktivasi berdasarkan suatu nilai *input*. Nilai *input* tersebut diperoleh dari hasil perhitungan jumlah *weighted sum* dari seluruh input neuron ditambah dengan suatu nilai bias apabila ada.

Fungsi aktivasi non-linier biasanya diterapkan setelah semua lapisan yang dapat dilatih dan memiliki bobot seperti *fully-connected layers* dan lapisan konvolusi pada arsitektur CNN. Karakteristik non-linier ini memungkinkan suatu jaringan untuk mempelajari pola-pola kompleks. Berikut adalah beberapa fungsi aktivasi non-linier yang sering digunakan.

1. Sigmoid: *Input* dari sigmoid adalah angka *real*, sedangkan *output* yang dikeluarkan berada pada rentang (0, 1). Sigmoid dapat direpresentasikan dalam bentuk matematis pada persamaan 2.3.

$$f(x)_{\text{sigm}} = \frac{1}{1+e^{-x}} \tag{2.3}$$

2. TanH: TanH serupa dengan fungsi aktivasi sigmoid, namun *output* yang dihasilkan berada pada rentang (-1, 1). TanH dapat direpresentasikan dalam bentuk matematis pada persamaan 2.4.

$$f(x)_{\text{tanh}} = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.4}$$

3. ReLU: ReLU mengubah *input* bernilai negatif menjadi 0 dan mempertahankan *input* yang bernilai positif. ReLU dapat dipresentasikan dalam bentuk matematis pada persamaan 2.5.

$$f(x) = \max(0,x) \tag{2.5}$$

Persamaan ini menunjukkan bahwa ReLU tidak akan pernah memiliki *output* yang bernilai negatif.

2.3.5 *Loss Function*

Loss function biasanya digunakan pada *output layer* untuk mengkalkulasi nilai *error* pada prediksi dibandingkan dengan nilai aktual suatu sampel data [20]. Nilai ini akan di optimisasi pada saat pelatihan CNN, di mana semakin rendah nilai *loss*, maka performa model dapat dikatakan lebih baik. Berikut adalah beberapa contoh *loss function*.

Salah satu *loss function* yang sering digunakan adalah *softmax*. *Softmax* biasa digunakan untuk mengukur performa model CNN. *Output* yang dihasilkan *softmax* memiliki rentang (0, 1) dan memiliki representasi matematis dalam persamaan 2.6.

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e_k^a} \tag{2.6}$$

di mana e^{a_i} merupakan *output* yang belum dinormalisasikan dari lapisan sebelumnya dan N merupakan jumlah neuron pada lapisan *output*.

2.3.6 Regularisasi

Regularisasi merupakan suatu teknik yang digunakan dalam *neural network* untuk mengurangi kemungkinan model untuk mengalami *over-fitting* dan *under-fitting* [20]. *Over-fitting* merupakan kasus di mana model memiliki performa yang baik pada data *training*, namun performa buruk pada data *validation* dan *testing*. Di sisi lain, *under-fitting* merupakan kasus di mana suatu model tidak belajar dari data *training*. Berikut adalah beberapa teknik regularisasi yang sering digunakan.

1. *Dropout*: Teknik ini menonaktifkan beberapa neuron secara acak pada setiap *training epoch*. Dengan melakukan ini, kemampuan *feature selection* didistribusikan secara merata kepada seluruh kelompok neuron yang ada. Maka, pada *training*, neuron yang dinonaktifkan tidak akan digunakan dalam *back-propagation* atau *forward-propagation*.

2. *Data Augmentation*: Teknik ini menghasilkan data *training* baru dengan melakukan beberapa transformasi pada data *training* awal. Dalam *image processing*, hal ini dapat meliputi *flipping*, *rotate*, *crop*, *brightness*, dan *distortion*. Dengan teknik ini, model dapat belajar fitur-fitur yang lebih *robust* dan *invariant* terhadap berbagai transformasi data.

3. *L2 Regularization*: Teknik ini menambahkan penalti terhadap bobot model dengan menambahkan jumlah kuadrat dari semua bobot ke dalam *loss function*. Dalam konteks *kernel learning*, regularisasi L2 digunakan untuk memilih kombinasi *kernel* yang optimal dari sejumlah *kernel* dasar dengan menjaga stabilitas model dan menghindari *overfitting* [23].

2.4 *Adaptive Moment Estimation* (ADAM)

ADAM adalah teknik optimisasi yang sering digunakan dalam pelatihan model *deep-learning*. Adam bekerja dengan mengkalkulasi nilai *learning rate* (LR) secara adaptif untuk tiap parameter pada *model* [20]. Metode ini menyimpan estimasi *exponentially weighted moving average*, yang masing-masing mewakili *mean* dan *variance* gradien yang belum terkoreksi. Kedua estimasi ini kemudian dikoreksi bias untuk menghasilkan estimasi yang tidak bias terhadap nol, sebelum

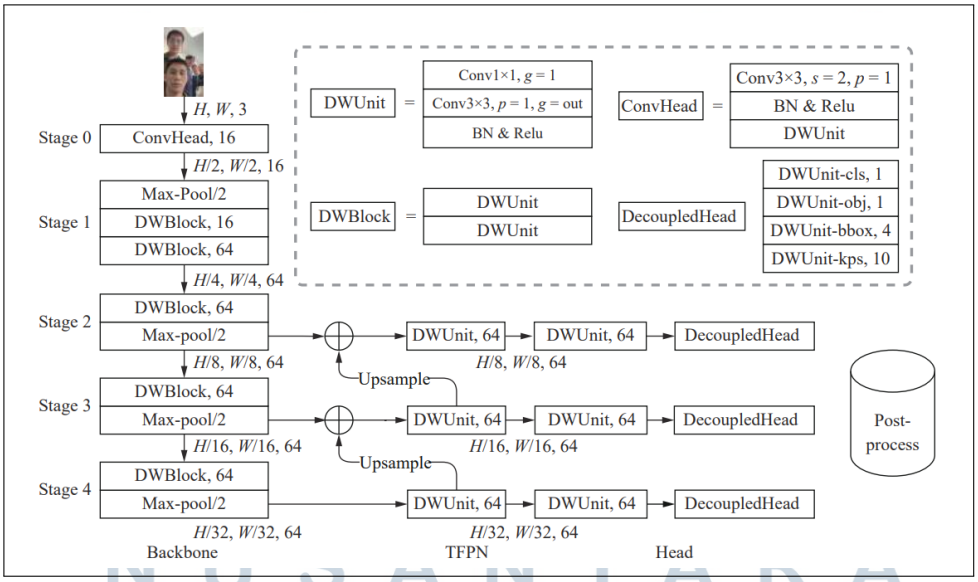
digunakan untuk memperbarui parameter model. Persamaan matematis ADAM dapat dipresentasikan pada persamaan 2.7.

$$w_{ij^t} = w_{ij^{t-1}} - \frac{\eta}{\sqrt{\widehat{E[\delta^2]^t} + \varepsilon}} \cdot \widehat{E[\delta^2]^t} \tag{2.7}$$

2.5 YuNet

YuNet adalah sebuah detektor wajah ringan berbasis CNN yang dirancang khusus untuk *edge devices*, yaitu perangkat keras yang melakukan *on-device processing* tanpa mengirim data ke *server* atau *cloud* [24]. Detektor wajah ini mengadopsi pendekatan *anchor-free* dan dirancang untuk memberikan efisiensi tinggi dalam inferensi tanpa mengorbankan akurasi secara signifikan.

YuNet mampu mencapai deteksi wajah dalam tingkat milidetik dengan jumlah parameter yang sangat rendah sejumlah 75856 *parameter*, dan memiliki keperluan komputasi yang jauh lebih rendah dibandingkan detektor wajah konvensional lainnya. Arsitektur YuNet terdiri dari tiga komponen utama, yaitu *backbone*, *tiny feature pyramid network* (TPFN) *neck*, dan *head* seperti pada gambar 2.7.



Gambar 2.7. Arsitektur YuNet

Sumber: [24]

2.5.1 *Backbone*

Komponen utama dari arsitektur jaringan YuNet adalah *backbone* yang dirancang untuk mengekstraksi fitur secara efisien dengan beban komputasi yang rendah, sehingga cocok untuk diterapkan pada *edge-devices*. Untuk mencapai efisiensi ini, YuNet menggunakan teknik konvolusi yang disebut *depthwise separable convolution* (DSC) [24]. Teknik ini memisahkan proses konvolusi menjadi dua tahap, yaitu *depthwise convolution* dan *pointwise convolution*. Meskipun teknik ini dapat menurunkan sedikit akurasi, beban komputasi dan jumlah parameter yang digunakan dapat berkurang dari 1/9 hingga 1/8 dari konvolusi standar 3x3.

Unit utama arsitektur ini, DWUnit, terdiri dari satu lapisan DSC, diikuti dengan *batch normalization* dan *activation function* [24]. Dua DWUnit membentuk satu DWBlock dan kedua modul ini merupakan blok dasar dalam desain jaringan YuNet. Struktur *backbone* terdiri dari lima tahap. Tahap pertama atau *stage* 0 menggunakan konvolusi standar dengan ukuran kernel 3x3 dan stride 2. Tahap ini dilanjutkan dengan *stage* 1, di mana diterapkan *maxpooling*, sebuah *RelU*, dan dua DWBlocks. Kedua tahap ini mengurangi *feature maps* menjadi 1/4 dari ukuran masukan dan meningkatkan jumlah kanal dari 3 menjadi 64. Tiga tahapan terakhir, *stage* 2 sampai 4, memiliki struktur jaringan yang sama dan menghasilkan fitur berlapis yang kemudian diteruskan ke bagian TPFN untuk pemrosesan lebih lanjut.

2.5.2 *Neck*

Bagian *neck* dari jaringan YuNet berperan dalam menggabungkan fitur dari berbagai skala untuk membentuk representasi fitur tingkat tinggi. Pada YuNet, bagian ini diimplementasikan dengan pendekatan bernama *tiny feature pyramid network* (TPFN), yang merupakan pengembangan dari arsitektur *feature pyramid network* (FPN) dengan mengintegrasikannya dengan DSC [24].

Tahap ketiga, *stage* 2, memiliki *feature map* yang bersifat *low-level*, sedangkan tahap kelima, *stage* 4, memiliki *high-level features* karena kedalaman jaringan yang berbeda. FPN sendiri memperkenalkan jalur *top-down* dan sambungan lateral untuk menggabungkan *multiscale faetures*.

Jalur *top-down* melakukan proses *upsampling* untuk meningkatkan resolusi fitur dari level-level atas, sedangkan sambungan lateral menggabungkan fitur dari *backbone* dan jalur *top-down* yang memiliki ukuran spasial yang sama.

Penggabungan fitur tersebut dapat diformulasikan menjadi persamaan 2.8.

$$P_i = F_i\left(R_{\text{dim}}(C_i) + R_{\text{size}}(C_{i+1})\right) \tag{2.8}$$

Proses ini terdiri atas tiga komponen, yaitu R_{dim} yang merupakan proses penyesuaian dimensi kanal, R_{size} yang merupakan proses *resolution matching*, dan F_i yang merupakan pemrosesan fitur melalui konvolusi.

Walaupun efektif, penggunaan konvolusi standar dalam FPN mengakibatkan tingginya jumlah parameter dan beban komputasi. Maka dari itu, YuNet menggunakan TPFN yang menggantikan konvolusi 3x3 dalam FPN dengan DWUnit, yaitu blok konvolusi ringan yang juga digunakan pada bagian *backbone*. Perubahan tersebut dapat mengurangi jumlah parameter hingga 12% dibandingkan FPN dan mengurangi beban komputasi secara signifikan.

2.5.3 *Head*

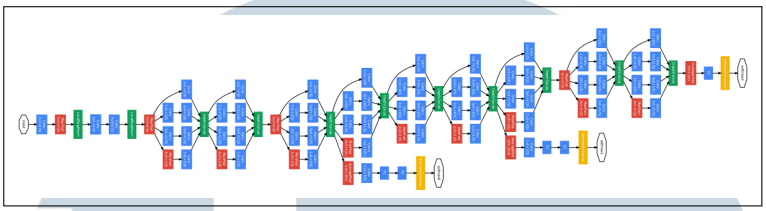
Bagian *head* dalam arsitektur YuNet menggunakan pendekatan *anchor-free* untuk mendeteksi objek (wajah) [24]. Pendekatan *anchor-free* berarti bahwa model tidak lagi bergantung pada *anchor boxes* yang telah ditentukan sebelumnya, melainkan langsung memprediksi koordinat objek dari setiap lokasi pada *feature map*. Mekanisme ini mengurangi rasio jumlah kandidat prediksi sebesar 2 banding 1 dan langsung memprediksi 4 nilai lokasi, yaitu 2 nilai untuk koordinat titik kiri atas, tinggi, dan lebar kotak pembatas.

Proses *positive anchor matching* antara prediksi dan *ground truth* menggunakan metode *simple optimal transport assignment* (simOTA). Pada tahap pencocokan ini, nilai *Intersection over Union* (IOU) antara prediksi dan *ground truth* sebagai *soft label*. Lalu dalam pelatihan digunakan *loss function* yang berupa *CrossEntropyLoss*. Sementara itu, regresi kotak pembatas memanfaatkan *Extended IoU Loss* (EIoU), regresi titik wajah menggunakan *Smooth1Loss*, dan prediksi keberadaan objek dihitung kembali dengan *CrossEntropyLoss*.

$$L = \frac{1}{N}\sum^{N}\left(L_{cls} + \alpha_1 L_{obj} + \alpha_2 L_{bbox} + \alpha_3 L_{ldm}\right) \tag{2.9}$$

Bagian *head* dilatih dengan tujuan untuk meminimalisir nilai *loss* yang terdapat pada persamaan 2.9 dengan N berupa jumlah total sampel positif dan nilai *hyperparameter* α_1, α_2, dan α_3 masing-masing direkomendasikan berada pada 1.0, 5.0, dan 0.1.

2.6 GoogleNet (InceptionV1)

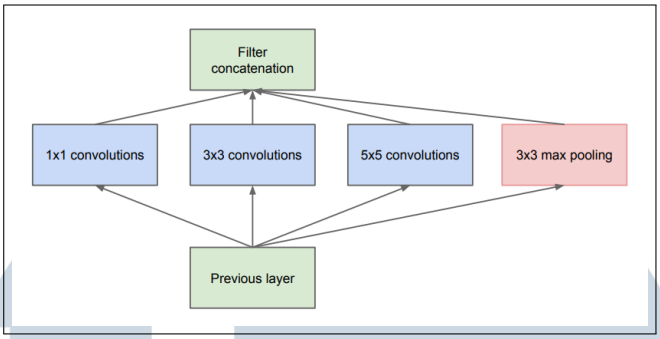


Gambar 2.8. InceptionV1 atau GoogleNet

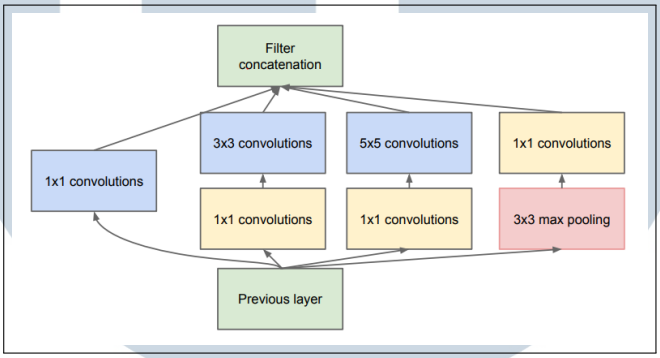
Sumber: [25]

GoogleNet, atau InceptionV1 dengan struktur seperti pada gambar 2.8, diperkenalkan pada tahun 2014 sebagai pemenang kompetisi *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) [25]. Arsitektur ini memperkenalkan konsep *Inception Module*, yaitu blok jaringan yang menjalankan operasi konvolusi paralel dengan ukuran *kernel* yang berbeda (1×1, 3×3, 5×5) dan lapisan *max-pooling*, lalu menggabungkan *output*-nya secara *depth-wise* seperti pada gambar 2.9.

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA



((a)) *Inception Module, Naïve Version*



((b)) *Inception Module with Dimensionality Reduction*

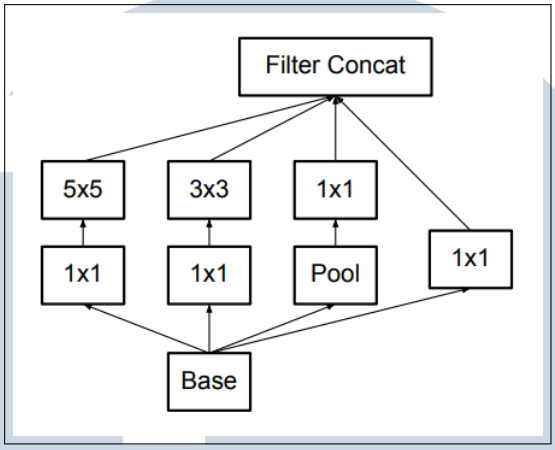
Gambar 2.9. Struktur *Inception Module*

Sumber: [25]

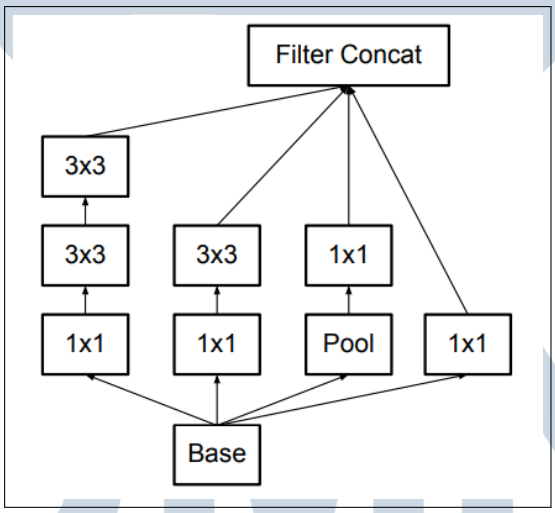
Inovasi utamanya terletak pada penggunaan konvolusi 1×1 sebagai *bottleneck* untuk mengurangi dimensi *channel* sebelum operasi konvolusi *kernel* besar, sehingga menekan kompleksitas komputasi dan parameter secara signifikan. Dengan hanya 5 juta parameter, GoogleNet lebih efisien dibandingkan arsitektur sebelumnya seperti AlexNet (60 juta parameter) dan VGGNet (138 juta parameter). Arsitektur ini menjadi fondasi untuk pengembangan Inception generasi selanjutnya.

UNIVERSITAS MULTIMEDIA NUSANTARA

2.7 InceptionV2



Gambar 2.10. *Old Inception Module*



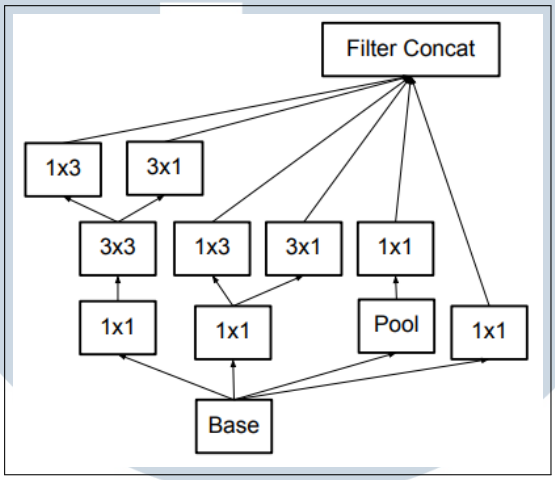
Gambar 2.11. *New Inception Module* dengan Pemecahan Konvolusi Besar

Setelah kesuksesan GoogleNet (InceptionV1), dikembangkanlah InceptionV2 yang bertujuan untuk mengatasi beberapa keterbatasan arsitektur sebelumnya. Salah satu perubahan besar pada InceptionV2 adalah pengenalan konsep *factorized convolutions*, di mana operasi konvolusi besar seperti 5×5 seperti pada gambar 2.10 dipecah menjadi dua konvolusi berturut-turut berukuran 3×3 seperti pada gambar 2.11 [26]. Pendekatan ini secara signifikan mengurangi jumlah parameter dan beban komputasi tanpa mengurangi kapasitas representasi fitur.

InceptionV2 juga memperkenalkan penggunaan lebih sistematis dari *batch normalization* dalam seluruh jaringan. *Batch normalization* membantu

mempercepat konvergensi saat pelatihan dan meningkatkan stabilitas jaringan dengan mengurangi variasi distribusi fitur internal selama pembelajaran. Dengan penggabungan inovasi-inovasi ini, InceptionV2 menawarkan efisiensi yang lebih baik dan performa akurasi yang lebih tinggi dibandingkan pendahulunya.

2.8 InceptionV3

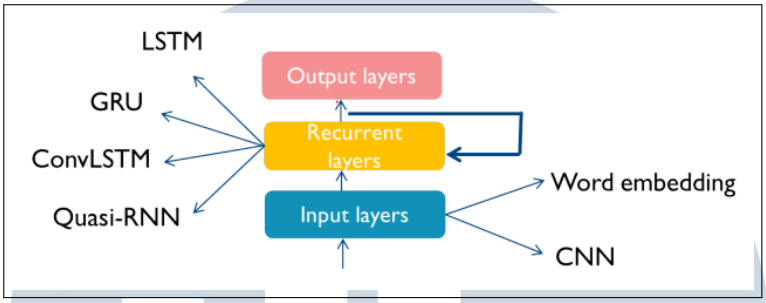


Gambar 2.12. *Inception Module* dengan *Factorized Convolution*

InceptionV3 merupakan generasi ketiga dari perkembangan arsitektur GoogleNet dan merupakan kelanjutan langsung dari inovasi yang diperkenalkan dalam InceptionV2. InceptionV3 juga menjadi *runner-up* pada kompetisi *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) 2015 [12]. Arsitektur ini lebih lanjut memperdalam konsep *factorized convolutions* dengan memperkenalkan konvolusi asimetris, seperti pemisahan konvolusi 3x3 menjadi operasi berturut-turut 1×3 dan 3×1 seperti pada gambar 2.12, yang semakin mengurangi beban komputasi tanpa mengurangi performa [26].

Selain itu, InceptionV3 mengadopsi regularisasi tambahan berupa *label smoothing* untuk memperbaiki kemampuan generalisasi model dengan mencegah prediksi menjadi terlalu yakin pada satu kelas. Blok-blok *Inception Module* dalam InceptionV3 juga mengalami penyempurnaan menjadi beberapa tipe khusus seperti *Inception-A*, *Inception-B*, dan *Inception-C* yang didesain untuk menangani perubahan dimensi *feature map* secara lebih efisien.

2.9 *Recurrent Neural Network* (RNN)



Gambar 2.13. *Recurrent Neural Network Layers*

Sumber: [27]

Recurrent Neural Network (RNN) adalah salah satu arsitektur jaringan saraf tiruan yang dirancang untuk mengolah data yang bersifat sekuensial [27]. Berbeda dari jaringan saraf *feedforward* yang hanya memiliki aliran data satu arah, RNN memiliki *feedback* yang memungkinkan informasi dari *timeframe* sebelumnya untuk digunakan dalam proses perhitungan dalam frame-frame selanjutnya. Hal ini memungkinkan jaringan RNN untuk menangkap hubungan temporal pada data *input* dan *output*. RNN memiliki tiga lapisan utama, yaitu *input*, *recurrent*, dan *output* seperti pada gambar 2.13.

2.9.1 *Input Layer*

Input layer bertugas untuk mempersiapkan *sensor output* untuk pemrosesan data atau *feature extraction*. Hal ini dikarenakan biasanya data *input* mentah seperti *video frame* biasanya belum sesuai untuk diproses secara langsung pada lapisan *recurrent* [27]. Selain itu, performa jaringan RNN, baik dalam *learning rate* ataupun akurasi, dapat ditingkatkan apabila fitur-fitur penting sudah diekstraksi dalam lapisan ini.

Dalam pemrosesan *input* berupa video, jaringan CNN umumnya digunakan pada *input layer* untuk mengekstraksi fitur-fitur penting dari setiap *frame* dalam rangkaian gambar video. Pendekatan ini telah banyak diaplikasikan dalam berbagai kasus, seperti *activity recognition*, *image description*, atau *video description*. Fitur-fitur relevan yang dihasilkan dari proses ekstraksi ini selanjutnya akan diteruskan ke lapisan *recurrent* untuk dianalisis secara temporal.

2.9.2 *Recurrent Layer*

Recurrent layer merupakan komponen utama dari RNN yang memungkinkan pemrosesan data sekuensial dengan memanfaatkan mekanisme *feedback*, yaitu keluaran dari suatu *time-step* $(t-1)$ dapat digunakan kembali sebagai *input* pada *time-step* berikutnya (t) [27]. Hal ini memungkinkan RNN untuk menyimpan konteks dari *input* sebelumnya dan menggunakannya dalam menghasilkan *output* saat ini, menjadikannya sangat efektif dalam tugas-tugas *language modelling*, *activity recognition*, dan *sentiment analysis*.

Secara teknis, lapisan *recurrent* menggabungkan vektor masukkan saat ini (x_t) dengan *hidden output* sebelumnya (h_{t-1}), kemudian memprosesnya menggunakan operasi linear (seperti perkalian matriks-vektor). penambahan bias, dan *linear activation* seperti *signmoid* dan *tanh*. Hasilnya adalah vektor *output* baru (h_t) yang tidak hanya mencerminkan informasi dari *input* saat ini, namun juga dari konteks historis sebelumnya.

2.9.3 *Output Layer*

Di dalam arsitektur RNN, *output layer* terdiri dari dua komponen utama, yaitu *Fully Connected Layer* dan *Output Function* [27].

A *Fully Connected (FC) Layer*

FC *layer* pada RNN berfungsi untuk mentransformasikan vektor tersembunyi (h_t) yang dihasilkan lapisan *recurrent* ke dalam bentuk sesuai untuk diproses oleh *output function* [27]. Hal ini dilakukan dengan perkalian antara matriks dengan vektor menggunakan bobot matriks berdimensi $Hidden_{size} \times Output_{size}$, di mana $Hidden_{size}$ merupakan ukuran vektor tersembunyi h_t dari lapisan *recurrent* dan $Output_{size}$ merupakan vektor *output*.

Model RNN dapat memiliki satu atau lebih lapisan FC setelah lapisan *recurrent*, dan dapat pula menerapkan fungsi non-linear di antara lapisan FC tersebut untuk meningkatkan kapasitas representasi dan kompleksitas model dalam menangkap pola-pola non-linear dalam data. Salah satu tujuan utama lapisan FC adalah untuk mengubah dimensi vektor tersembunyi h_t agar sesuai dengan dimensi *output* yang diharapkan untuk digunakan oleh *output function*. Dalam beberapa arsitektur, fungsi transformasi ini juga dapat digantikan dengan menambahkan lapisan *projection* yang tertanam langsung pada lapisan *recurrent* [27].

B *Output Function*

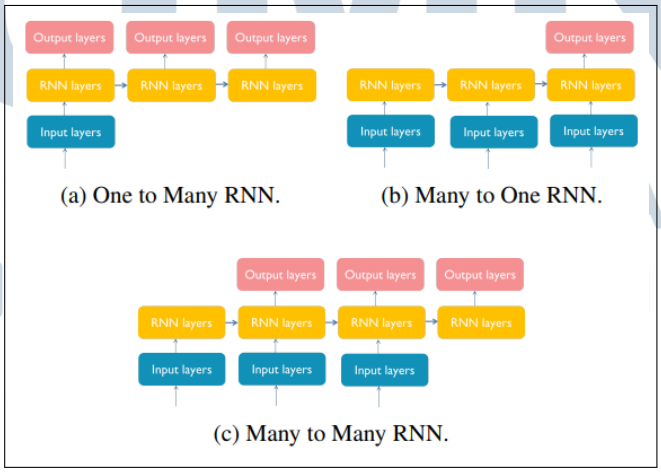
Dalam proses inferensi *neural network*, *output function* merupakan tahap terakhir yang menghasilkan *output* dari jaringan tersebut [27]. *Output* tersebut dapat berupa berbagai hal seperti prediksi, klasifikasi, pengenalan, dan sebagainya, bergantung pada konteks penggunaannya. Fungsi ini bekerja dengan mengubah hasil dari lapisan FC menjadi suatu vektor probabilitas, di mana seluruh elemen dalam vektor tersebut dijamin memiliki jumlah total berupa satu.

Salah satu contoh umum dari fungsi ini adalah *softmax*, yang sering digunakan dalam tugas klasifikasi. Fungsi softmax menghitung eksponensial untuk setiap elemen vektor skor, lalu melakukan normalisasi dengan membagi setiap nilai eksponensial dengan jumlah total semua nilai eksponensial tersebut. Dengan demikian, setiap elemen dalam vektor output mewakili probabilitas relatif dari masing-masing kelas, dan kelas dengan nilai probabilitas tertinggi dipilih sebagai prediksi akhir jaringan [27].

C Pemrosesan Data pada Model RNN

Pemrosesan data dalam arsitektur RNN dapat berbeda-beda, tergantung pada jenis aplikasi dan struktur temporal yang digunakan [27]. Secara umum, terdapat dua bentuk variasi utama dalam pemrosesan data pada RNN, yaitu variasi dalam pengolahan berdasarkan *time step* dan pemanfaatan RNN bidireksional.

1. **Variasi Berdasarkan Urutan Waktu**



Gambar 2.14. Pemrosesan Data pada Model RNN Berdasarkan Urutan Waktu
Sumber: [27]

Untuk memvisualisasikan bagaimana RNN bekerja melintasi waktu, dilakukan proses *unfolding* atau membuka jaringan sepanjang waktu. Proses ini menunjukkan pengulangan dalam lapisan *recurrent* dan menggambarkan berapa banyak *time-step* yang dibutuhkan untuk menjalankan suatu tugas. Berikut adalah berbagai tipe model RNN yang dapat ditemukan berdasarkan urutan waktu seperti pada gambar 2.14.

(a) ***One to Many***

Model *One to Many* menerima satu masukkan tunggal dan menghasilkan beberapa *output* berurutan. Contoh aplikasi model ini adalah *image captioning*, di mana sebuah gambar diambil sebagai masukkan dan sebuah kalimat dihasilkan yang kata-katanya terdiri dari data temporal terkait yang berurutan. Dalam jenis model ini, hubungan temporal hanya terdapat pada sisi *output*.

(b) ***Many to One***

Model *Many to One* bekerja dengan menerima serangkaian masukkan secara berurutan dan menghasilkan satu *output* tunggal. Contoh aplikasi model ini adalah *activity recognition* dan *sentiment analysis*. Kedua aplikasi ini mengambil berbagai masukkan untuk menentukan satu buah *output*. Dalam jenis model ini, hubungan temporal hanya ada di sisi *input*.

(c) ***Many to Many***

Model *Many to Many* memiliki urutan baik pada masukkan maupun keluaran. Contohnya adalah pada aplikasi penerjemah bahasa dan deskripsi video. Kedua aplikasi tersebut akan mengambil banyak *input* dan menghasilkan banyak *output*.

(d) ***One to One***

Model *One to One* tidak memiliki hubungan temporal baik pada *input* maupun *output* atau dapat disebut sebagai *feedforward neural network*.

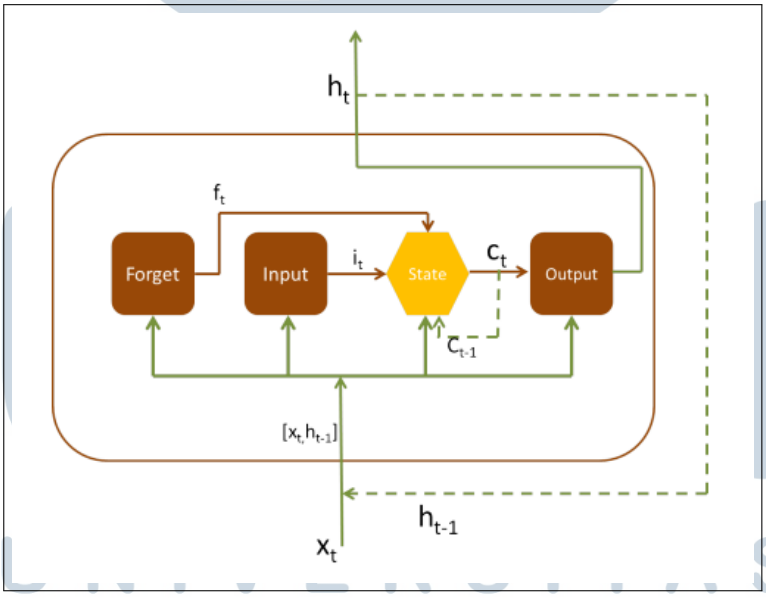
2. ***Bidirectional RNN***

Dalam arsitektur *Bidirectional* RNN, data *input* diproses tidak hanya dari masa lalu ke masa depan, tapi juga dari masa depan ke masa lalu [27]. Hal ini dilakukan dengan menduplikasi lapisan *recurrent*, sehingga terdapat dua lapisan yang berjalan secara paralel namun berlawanan arah dalam konteks urutan waktu.

Tujuan dari pendekatan ini adalah untuk memungkinkan jaringan memperoleh konteks yang lebih lengkap, baik dari data sebelumya maupun data selanjutnya, secara bersamaan. Strategi ini sangat bermanfaat dalam aplikasi yang memerlukan pemahaman menyeluruh terhadap konteks temporal secara *bidirectional*. Konsep ini juga dapat diterapkan pada berbagai jenis lapisan *recurrent*, seperti *Bidirectional LSTM* dan *Bidirectional GRU*.

2.10 *Long Short-Term Memory* (LSTM)

Long Short-Term Memory atau LSTM merupakan salah satu varian dari lapisan *recurrent* dari RNN yang dibuat oleh Hochreiter dan Schmidhuber pada tahun 1997. LSTM dirancang untuk menangani masalah *long-term dependencies* dalam data sekuensial. Secara umum, LSTM menerima masukan berupa gabungan dari vektor input x_t, dan keluaran dari waktu sebelumnya h_{t-1}, kemudian menghasilkan keluaran baru h_t [27].



Gambar 2.15. Arsitektur LSTM
Sumber: [27]

Di dalam arsitektur LSTM seperti pada gambar 2.15, terdapat suatu ”*cell state*” C_t yang berfungsi sebagai memori jangka panjang, serta tiga gerbang utama: *forget gate*, *input gate*, dan *output gate*. Ketiga gerbang ini mengatur informasi mana yang harus dilupakan, diperbarui, atau digunakan sebagai *output*.

Dari sisi kompleksitas komputasi, LSTM sangat bergantung pada operasi perkalian matriks dengan vektor. Untuk setiap *time-state*, empat kali operasi semacam ini dilakukan, masing-masing dengan kompleksitas $n(m+n)$, di mana m merupakan ukuran *input* dan n merupakan ukuran *state* tersembunyi. Karena biasanya $n > m$, parameter n memiliki pegnaruh dominan terhadap jumlah total operasi dan parameter yang dibutuhkan oleh model LSTM [27].

Computation block LSTM adalah matrix dari perkalian vektor antara kombinasi dari vektor *input* dan vektor *output* dengan salah satu matriks bobot $\{W_f, W_i, W_c, W_o\}$ [27]. Proses ini dilanjutkan dengan penambahan vektor bias $\{b_f, b_i, b_c, b_o\}$ dan penerapan fungsi aktivasi non-linier (signmoid atau tanh). Setiap blok juga dapat mencakup operasi perkalian *element-wise*.

2.10.1 *Forget Gate*

Forget gate bertugas untuk memutuskan informasi mana dari *state* sebelumnya yang perlu dilupakan. *Output* dari *forget gate* f_t dihitung dengan persamaan 2.10

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{2.10}$$

di mana x_t merupakan vektor *input*, h_{t-1} merupakan *hidden state* vektor *output*, W_f merupakan matriks bobot, b_f merupakan vektor bias, dan σ merupakan fungsi *sigmoid*.

2.10.2 *Input Gate*

Input gate bertugas untuk memutuskan informasi mana dari *state* sebelumnya yang perlu diperbarui. *Output* dari *input gate* dihitung dengan persamaan 2.11

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.11}$$

di mana x_t merupakan vektor *input*, h_{t-1} merupakan *hidden state* vektor *output*, W_i merupakan matriks bobot, b_i merupakan vektor bias, dan σ merupakan fungsi *sigmoid*.

2.10.3 *State Computation*

State computation bertugas untuk menghitung *memory state* baru C_t dari suatu *cell* LSTM. Pertama, nilai kandidat dihitung dengan persamaan 2.12

$$\tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{2.12}$$

di mana x_t merupakan vektor *input*, h_{t-1} merupakan *hidden state* vektor *output*, W_c merupakan matriks bobot, b_c merupakan vektor bias. Lalu, C_t dikalkulasi dengan penambahan antara vektor *state* C_{t-1} yang dikalikan secara *element-wise* dengan vektor *output forget gate* f_t dengan kandidat baru $\tilde{C}_t$ yang dikalikan secara *element-wise* dengan vektor *output input gate* i_t seperti pada persamaan 2.13

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{2.13}$$

di mana $\odot$ digunakan untuk menandakan perkalian *element-wise*.

2.10.4 *Output Gate*

Fungsi *output gate* adalah menghasilkan *output* dari LSTM. Pertama, vektor *output gate* dikalkulasi dengan persamaan 2.14

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2.14}$$

di mana x_t merupakan vektor *input*, h_{t-1} merupakan *hidden state* vektor *output*, W_o merupakan matriks bobot, b_o merupakan vektor bias, dan σ merupakan fungsi *sigmoid*. Lalu, *hidden state output* h_t dikalkulasi dengan mengalikan vektor *output gate* o_t yang memutuskan bagian *state* yang merupakan *output* secara *element-wise* dengan *tanh* dari vektor *state* C_t seperti pada persamaan 2.15

$$h_t = o_t \odot tanh(C_t) \tag{2.15}$$

UNIVERSITAS MULTIMEDIA NUSANTARA

2.11 Metrik Evaluasi Model

2.11.1 *Confusion Matrix*

Confusion matrix adalah sebuah matriks $N \times N$, dengan N sebagai jumlah kelas dalam model, yang menunjukkan berbagai jenis *error* yang dihasilkan oleh model, seperti *true positive* (TP), *false positives* (FP), *true negatives* (TN), dan false negatives (FN) [28]. Dalam suatu klasifikasi biner, maka *confusion matrix* akan berbentuk matriks 2×2 seperti pada tabel 2.3.

Tabel 2.3. *Confusion Matrix* Berukuran 2×2

Actual	Predicted	
	Positive	Negative
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Negative (TN)

Nilai-nilai dalam *confusion matrix*, yaitu TP, FP, TN, serta FN, dapat diturunkan untuk mengkalkulasi nilai-nilai metriks evaluasi model lainnya, yaitu *accuracy*, *precision*, *recall*, dan *F1-score*.

A *Accuracy*

Nilai akurasi A klasifikasi model merupakan rasio data yag diklasifikasikan dengan akurat. Nilai ini dapat ditentukan dengan membagi nilai jumlah prediksi yang akurat dengan jumlah total prediksi yang dilakukan oleh model seperti pada persamaan 2.16.

$$A = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{TN} + \mathrm{FN}} \tag{2.16}$$

B *Precision*

Nilai presisi P klasifikasi model merupakan rasio antara kasus positif yang diprediksi secara akurat terhadap total kasus positif yang diprediksi. Nilai ini dapat ditentukan dengan membagi jumlah kasus positif yang secara akurat diprediksi

dengan total kasus positif yang diprediksi seperti pada persamaan 2.17.

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.17}$$

C ***Recall***

Nilai *recall R* merupakan perbandingan antara jumlah kasus positif yang berhasil diobservasi dengan jumlah kasus positif sebenarnya. Nilai ini dapat ditentukan dengan membagi jumlah kasus positif dengan total kasus positif sebenarnya ($Tp + FN$) seperti pada persamaan 2.18. 2.17.

$$P = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.18}$$

D ***F1-Score***

Nilai *F1-score* merupakan metrik yang mengevaluasi akurasi model pada setiap kelas yang sering digunakan untuk *dataset* yang memiliki perbandingkan kelas yang tidak seimbang. Nilai ini dapat ditentukan dengan menghitung rata-rata harmonik antara presisi dan recall seperti pada persamaan 2.19.

$$P = 2 \times \frac{\text{PR}}{\text{P} + \text{R}} \tag{2.19}$$

2.11.2 *AUC (Area Under the Curve) & ROC (Receiver Operating Characteristic)*

Receiver Operating Characteristic atau ROC adalah pendekatan analitik yang digunakan untuk mengevaluasi kinerja diagnostik suatu model atau sistem klasifikasi [29]. ROC menggambarkan hubungan antara *True Positive Rate* atau TPR dan *False Positive Rate* atau FPR pada suatu *decision threshold*.

Dengan melakukan plotting nilai TPR dan FPR pada setiap *threshold*, ROC menciptakan kurva yang menunjukkan sejauh mana suatu model mampu

membedakan antara dua kelas, seperti data asli dan *deepfake* sebagai contoh. Persamaan 2.20 menentukan nilai TPR dan persamaan 2.21 menentukan nilai FPR.

$$TPF(TPR) = \frac{a}{a+c} \tag{2.20}$$

Dengan a sebagai jumlah *true positives* dan c sebagai jumlah *false negatives*

$$FPF(FPR) = 1 - \text{Specificity} = \frac{b}{b+d} \tag{2.21}$$

Dengan b sebagai jumlah *false positives* dan d sebagai jumlah *true negatives*

Area Under the Curve atau AUC di sisi lain merupakan bentuk interpretasi terhadap keseluruhan luas kurva ROC yang menggambarkan performa diskriminatif dari model [29]. Nilai AUC berkisar dari 0.5 yang tidak lebih baik dari tebakan *random* hingga 1.0 yang merupakan diskriminasi sempurna. Dalam aplikasi praktis, nilai AUC di atas 0.9 dianggap sangat baik, 0.8-0.9 cukup baik, sedangkan AUC di bawah 0.8 sering kali dianggap terbatas dalam kegunaan klinis ataupun signifikan secara statistik.

Interpretasi nilai AUC juga harus memperhatikan *confidence interval* 95%, karena interval yang sempit menunjukkan presisi tinggi, sedangkan interval yang lebar menandakan ketidakpastian yang lebih besar terhadap estimasi nilai AUC. Selain itu, analisis ROC dapat digunakan untuk menentukan *cutoff value* optimal dengan menggunakan *Youden Index*, yaitu metrik yang memaksimalkan jumlah sensitivitas dan spesifisitas dikurangi satu.

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA