# BAB III

# METODOLOGI PENELITIAN

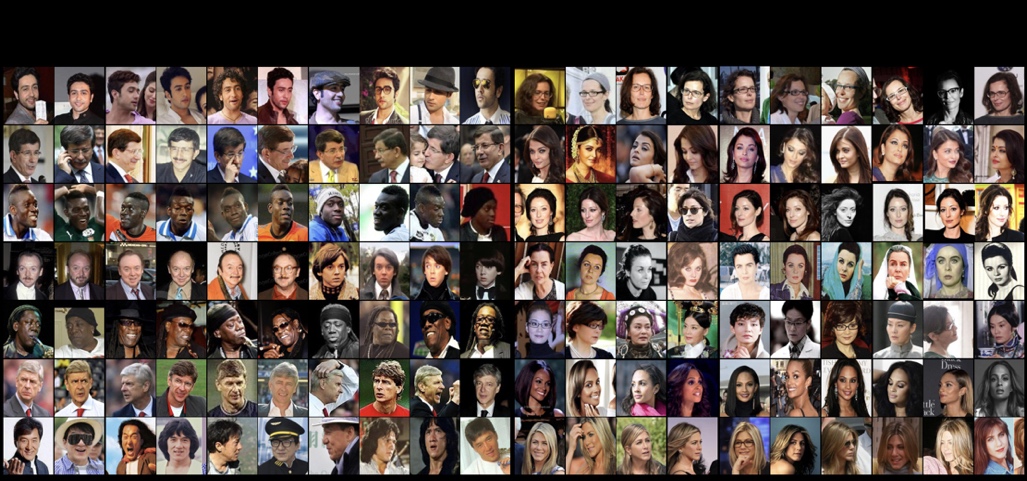
## Dataset

Pada penelitian ini, dataset yang akan digunakan sebagai masukan (*input*) kedalam arsitektur jaringan adalah gambar wajah yang valid, gambar wajah yang valid yang dibutuhkan peneliti adalah gambar wajah satu orang pada setiap gambar yang mewakili gambar wajah individu tersebut, jenis/tipe gambar yang digunakan adalah jpg 3 kanal(RGB)*.* Dataset yang digunakan peneliti diperoleh dari beberapa sumber terbuka seperti dijelaskan sebagai berikut:

* 1. **Data Latih (*Training*)**

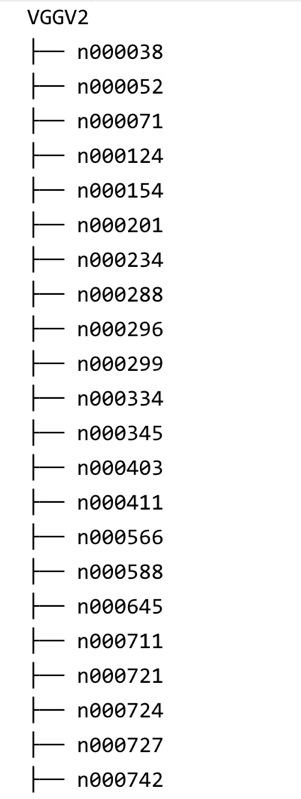
Data *training* yang digunakan dalam penelitian ini adalah VGG *Face* 2. VGG *Face* 2 adalah dataset *face recognition* dalam jumlah sangat besar, gambar-gambar pada dataset ini didownload dari pencarian gambar Google dan memiliki banyak variasi dalam hal pose, iluminasi, etnik dan profesi (Qiong Cao, Shen, Weidi Xie, Parkhi, & Zisserman, 2018). Dataset ini disebarkan secara bebas menggunakan lisensi *Creative Common Attribution-ShareAlike 4.0* namun hak cipta setiap gambar yang ada pada dataset ini tetap menjadi milik pemilik asal*.* VGG *Face* 2memiliki lebih dari 9000 jenis identitas dengan variasi etnis, aksen, profesi dan umur yang berbeda dan memiliki 3,3 juta gambar wajah dengan berbagai keadaan latar belakang, pose, ekspresi wajah dan pencahayaan yang berbeda-beda dengan rata-rata 362 gambar wajah pada setiap subjek (Qiong Cao et al, 2018). Grafik distribusi gambar dan contoh gambar pada dataset ini bisa dilihat pada gambar 3.1 dan gambar 3.2 dibawah.

|  |  |
| --- | --- |
| genderDistribusi Gender | train_valPembagian *Train/Test* |
| facesize  Distribusi Ukuran Gambar | |
| Gambar 3.1 Grafik Distribusi Dataset VGG *Face* v2[[1]](#footnote-1) | |



Gambar 3.2 Contoh Gambar Dataset VGG Face v2[[2]](#footnote-2)

Dataset VGG *Face* v2 disusun menjadi beberapa folder sesuai jumlah individu yang dijelaskan diatas, dimana setiap folder adalah label/*class*. Struktur folder pada dataset VGG *Face* v2 disusun seperti gambar 3.3 dibawah.



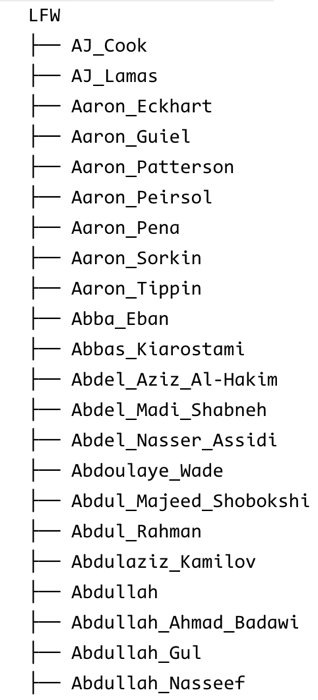
Gambar 3.3 Struktur Folder Dataset VGG *Face* v2

* 1. **Data Uji (*Testing*)**

Data *testing* yang akan digunakan pada penelitian ini adalah dataset LFW (*Labeled Face in the Wild*). Dataset ini disediakan oleh University of Massachusetts, motivasi utama dari dataset ini adalah untuk menyediakan gambar wajah yang bervariasi sesuai dengan yang ditemukan pada kehidupan sehari-hari dengan beragam variasi seperti pose, pencahayaan, ekspresi, latar belakang, ras, etnis, umur, jenis kelamin, pakaian, gaya rambut, kualitas kamera, saturasi warna, fokus gambar dan parameter lainnya (Huang, Ramesh, Berg, & Learned-Miller, 2008).

Dataset ini berisi 13,233 gambar, beberapa gambar ada yang memiliki lebih dari 1 wajah namun wajah yang akan diambil adalah wajah yang mendominasi bagian tengah gambar, yang lainnya akan dianggap sebagai latar belakang.

Dataset LFW memiliki 5749 jenis individu, 1680 diantaranya memiliki lebih dari 1 gambar dan sisanya (4069) hanya memiliki satu gambar, gambar tersedia dalam format JPEG 3 kanal warna dan ada beberapa gambar berwarna abu (*grayscale*). Gambar struktur folder dan contoh dataset LFW bisa dilihat pada gambar 3.4 dan 3.5 dibawah.

****

Gambar 3.4 Struktur Folder Dataset LFW

****

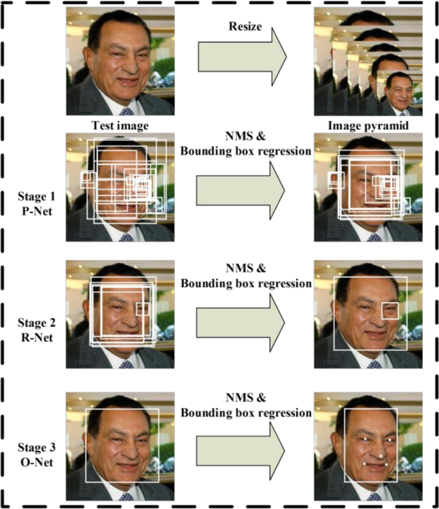
Gambar 3.5 Contoh Gambar Dataset LFW[[3]](#footnote-3)

* 1. **MTCNN**

Sebelum dataset digunakan untuk keperluan *training* maupun *testing* akan dilakukan persiapan terlebih dahulu. Persiapan ini sangat penting untuk memastikan dataset yang digunakan adalah dataset yang benar-benar valid sesuai kebutuhan training (poin 1 dataset). Adapun persiapan dataset yang digunakan pada penelitian ini adalah *Multi-task Cascaded Convolutional Networks (*MTCNN*).*

MTCNN adalah *state of the art (sota)* saat ini dalam mencari 5 titik wajah (*face landmark*) yaitu kedua mata (kiri dan kanan), hidung dan kedua ujung bibir samping kiri dan samping kanan(Zhang et al., 2016). MTCNN pada penelitian ini digunakan untuk memotong (krop) secara otomatis bagian wajah pada setiap gambar dataset dengan ukuran tertentu (penelitian ini menggunakan ukuran 182x182).

Adapun *framework* ini melalui 3 tahap CNN untuk melakukan proses pendeteksian *face landmark* seperti ditunjukan pada gambar 3.6 dibawah.



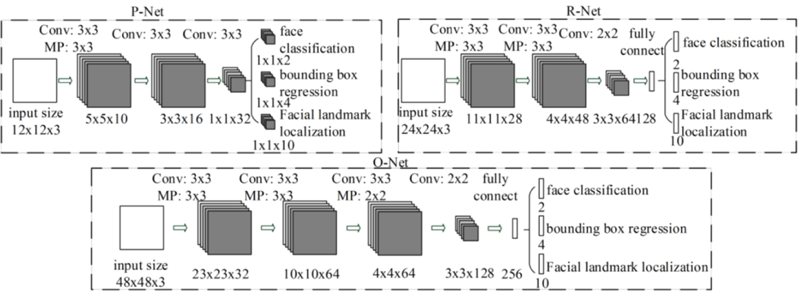
Gambar 3.6 *Pipeline* MTCNN Dalam Menghasilkan *Face Landmark*[[4]](#footnote-4)

Pada gambar 3.6 diatas gambar terlebih dahulu di-*resize* menjadi beberapa ukuran (*image pyramid*) sebelum diteruskan ke 3 buah tahap CNN yaitu:

**Tahap 1:** gambar yang telah di resize menjadi beberapa ukuran tertentu sebelumnya diteruskan ke arsitektur CNN yang disebut P-Net (*Proposal Network*). Fungsi dari jaringan ini adalah untuk mendapatkan beberapa *bounding box* (kotak batas sebagai acuan krop) lalu mengkalibrasinya agar mendapatkan hasil yang lebih cocok menggunakan estimated *bounding box regression vectors*, kemudian dari beberapa *bounding box* yang telah didapatkan dilakukan penggabungan *bounding box* menggunakan *Non-Maxima Suppression* (NMS) pada *bounding box* yang bertumpukkan dengan sangat rapat.

**Tahap 2:** setiap calon yang didapatkan pada tahap 1 kemudian diteruskan ke arsitektur CNN yang lain diberi nama *Refine Network* (F-Net). Fungsi dari jaringan ini adalah mengeliminasi calon yang salah (prediksi wajah yang tidak benar) lalu melakukan kalibrasi lagi dan melakukan proses NMS lagi.

**Tahap 3:** proses ini sama dengan tahap 2, namun pada tahap ini arsitektur jaringan ini akan mendeskripsikan lebih detail lagi *bounding box* yang didapatkan pada tahap 2 dan mengeluarkan *face landmark.* Arsitektur jaringan dari 3 tahap ini bisa dilihat pada gambar 3.7 dibawah.



Gambar 3.7 Arsitektur MTCNN[[5]](#footnote-5)

* 1. **Preprocessing**

Pada tahap *preprocessing,* data *training* maupun *testing* akan melalui beberapa perubahan (*transformasi*) agar dapat diterima dan bisa diproses sesuai desain arsitektur jaringan yang digunakan. Data *training* dan data *testing* akan melalui tahap proses transformasi yang berbeda sesuai yang dijelaskan pada poin-poin dibawah.

* + 1. Transformasi data *training*

Tahap transformasi pada *batch* data *training* akan melalui beberapa proses *transformasi* acak ini bertujuan untuk menambah variasi gambar (data augmentasi), beberapa *transformasi* acak yang dilakukan adalah sebagai berikut:

1. *Random Rotation*

Transformasi ini akan merotasi gambar secara acak sesuai sudut derajat yang ditentukan. Pada penelitian ini sudut derajat yang digunakan adalah 15°.

1. *Random Resized Crop*

Transformasi ini akan melakukan operasi krop pada gambar dengan ukuran yang bervariasi dari 8% sampai 100% persen dengan aspek rasio ¾ atau 4/3 dari ukuran gambar aslinya secara acak. Kemudian gambar yang telah dikrop akan di*resize* ke ukuran yang sudah ditentukan. Ukuran *resize* yang digunakan pada penelitian ini adalah 224 karena arsitektur jaringan yang digunakan membutuhkan gambar dengan ukuran 224x224 sebagai masukan.

1. *Random Horizontal Flip*

Transformasi ini berfungsi untuk membalik gambar pada arah horizontal secara acak. Proses pembalikan akan dilakukan dengan menentukan batas probabilitas gambar akan dibalik, lalu memilih angka secara acak diantara 0 sampai 1, jika angka acak yang didapatkan lebih kecil dari batas probabilitas yang telah ditentukan maka proses pembalikan horizontal akan dilakukan. Pada penelitian ini batas probabilitas yang digunakan adalah nilai *default* yaitu 0.5.

* + 1. Transformasi data *testing*

Transformasi pada data *testing* sama namun sedikit berbeda dengan tansformasi yang dilakukan pada data *training*, jika pada data *training* menggunakan beberapa transformasi acak untuk keperluan augmentasi data, pada data *testing* transformasi acak tidak dilakukan yang akan dilakukan hanyalah merubah (transformasi) data *testing* agar dapat diterima sebagai masukan pada arsitektur jaringan. Dibawah adalah transformasi yang dilakukan pada data *testing:*

1. *Resize*

Transformasi *resize* adalah proses merubah ukuran gambar dengan ukuran tertentu menggunakan algoritma interpolasi bilinear. Pada penelitian ini ukuran *resize* gambar yang digunakan adalah 224x224 sesuai kebutuhan arsitektur jaringan yang digunakan.

1. *Center Crop*

Sesuai namanya, proses transformasi ini akan memotong gambar pada posisi tengah dengan ukuran tertentu. Ukuran yang digunakan pada penelitian ini adalah 224x224 sesuai kebutuhan arsitektur jaringan yang digunakan.

* + 1. Normalisasi Gambar

Normalisasi data pada umumnya dilakukan agar setiap data berada pada skala distribusi tertentu. Namun pada kasus gambar digital sebenarnya normalisasi tidak dibutuhkan lagi (Andrej Karpathy, 2017) karena distribusi setiap pixel sudah berada pada rentang tertentu yaitu 0-255 atau biasa disebut skala keabuan, namun pada penelitian ini normalisasi data tetap dilakukan karena alasan spesifikasi arsitektur jaringan yang digunakan, dijelaskan lebih rinci pada sub bab 3.2.1 poin ii dibawah. Proses normalisasi data setiap pixel diperoleh melalui perhitungan *z-score* setiap pixel:

**Dimana:**

= hasil normalisasi.

= nilai pixel yang akan dinormalkan.

= nilai rata-rata.

= nilai simpangan baku (*standard deviation*).

Pada penelitian ini, nilai dan telah ditentukan sesuai ketentuan arsitektur jaringan yang digunakan sesuai tabel 3.1 dan tabel 3.2 dibawah.

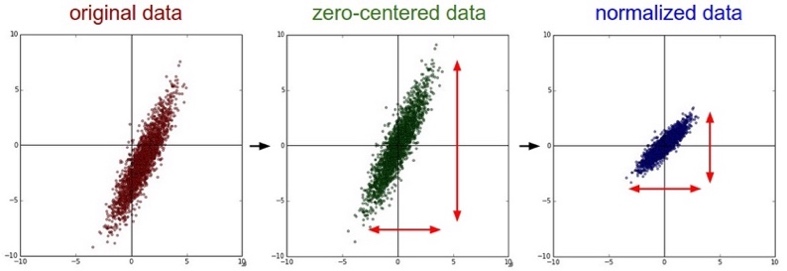
Tabel 3.1 Ketentuan Nilai *Mean* Yang Digunakan

|  |  |
| --- | --- |
| Nilai *mean* masing-masing kanal warna | |
| R | 0.485 |
| G | 0.456 |
| B | 0.406 |

Tabel 3.2 Ketentuan Nilai *Std* Yang Digunakan

|  |  |
| --- | --- |
| Nilai *std* masing-masing kanal warna | |
| R | 0.229 |
| G | 0.224 |
| B | 0.225 |

Persamaan diatas secara intuisi bisa dijelaskan seperti gambar 3.8 dibawah.



Gambar 3.8 Visualisasi Normalisasi Data[[6]](#footnote-6)

Komponen pertama pada persamaan diatas yaitu bertujuan untuk membuat setiap data *point* (nilai setiap pixel) terdistribusi secara relatif ke tengah. Lalu komponen kedua yaitu pembagian dengan nilai untuk mengurangi *skewness* (kemerongan)pada data sehingga panjang dan tinggi data terdistribusi secara merata seperti divisualisasikan pada gambar 3.8 paling kanan diatas.

* + 1. Transformasi Gambar Ke Tensor

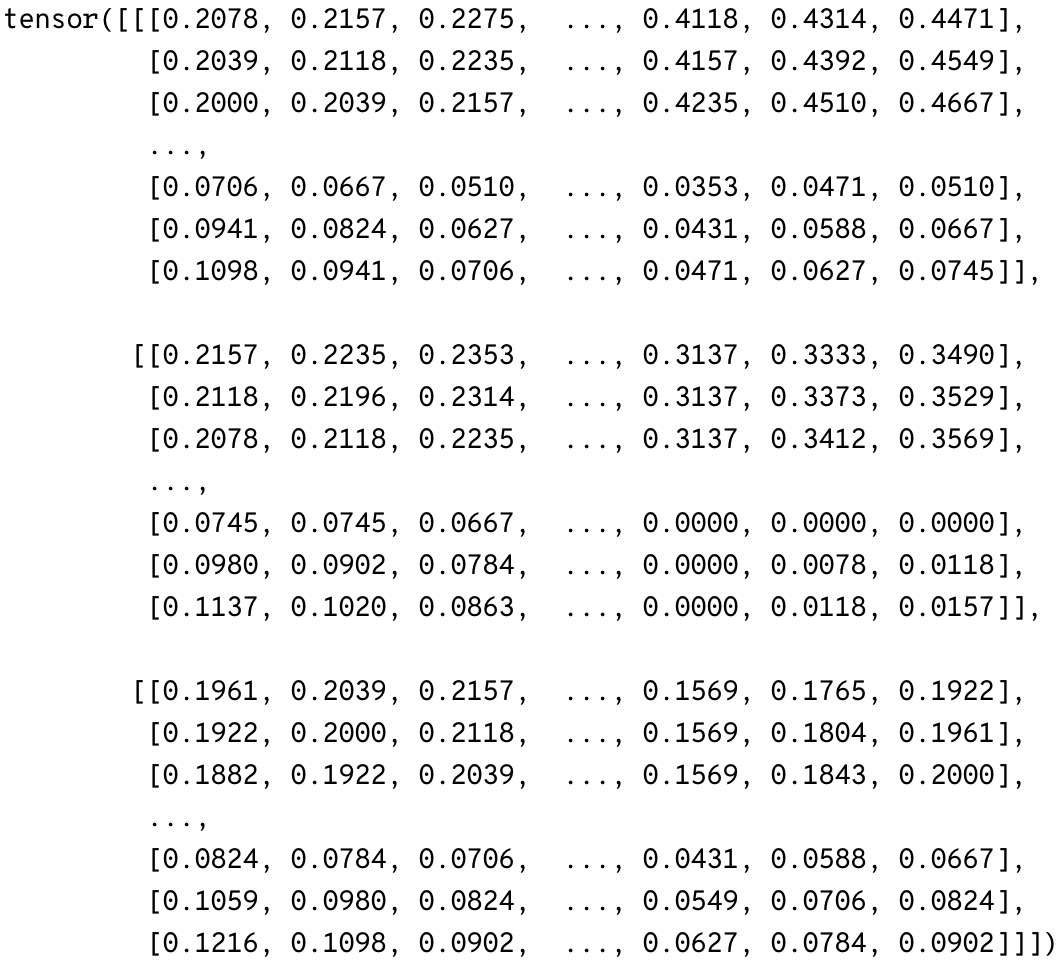
Tahap ini akan mengubah gambar menjadi *tensor* dengan nilai setiap elemen adalah nilai skala keabuan pada setiap pixel gambar. Ilustrasi hasil pemrosesan data pertahap bisa dilihat pada tabel 3.3 dan 3.4 dibawah dimulai dari gambar *raw* (gambar mentah) sebelum diproses kemudian dilanjutkan ke kanan sampai pada proses *data normalization,* hasil transformasi gambar ke tensor bisa dilihat pada gambar 3.9 dibawah.

Tabel 3.3 Ilustrasi *Preprocessing* Gambar Untuk *Training*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Raw input | MTCNN 182x182 | *Random Rotation* 15° | *Random Resized Crop* 224x224 | *Random Horizontal Flip* | Data *normalization* |
|  |  |  |  |  |  |

Tabel 3.4 Ilustrasi *Preprocessing* Gambar Untuk *Testing*

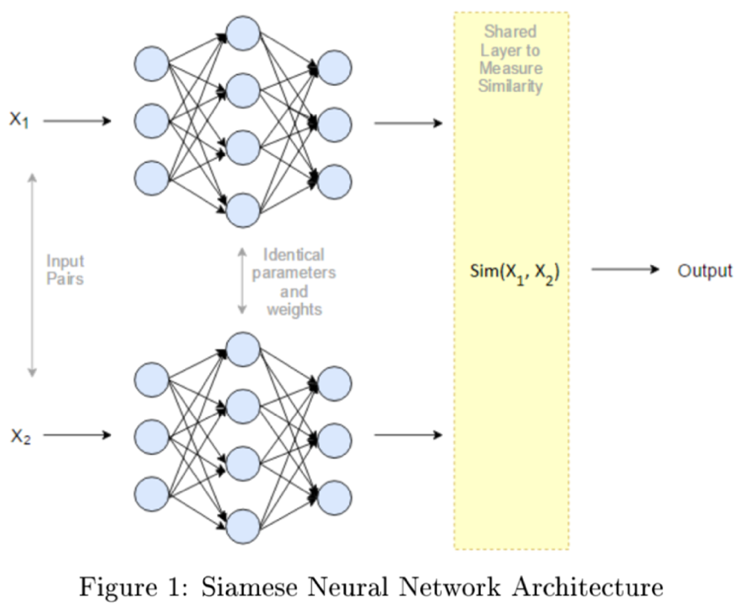
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Raw input | MTCNN 182x182 | *Resize* 224x224 | *Center Crop* 224x224 | Data *normalization* |
|  |  |  |  |  |



Gambar 3.9 Transformasi Gambar Ke Tensor

## Siamese

Arsitektur *siamese* pertama kali diperkenalkan oleh Jane Bromley, Yan LeCun et al pada tahun 1994. Arsitektur jaringan ini pada saat itu digunakan untuk menyelesaikan masalah verifikasi tanda tangan dengan capaian akurasi sampai 95%. Konsep utama dari jaringan ini adalah agar melewatkan pasangan *input* melalui jaringan yang identik (memiliki nilai parameter yang sama). Contoh arsitektur jaringan *Siamese* bisa dilihat pada gambar 3.10 dibawah.



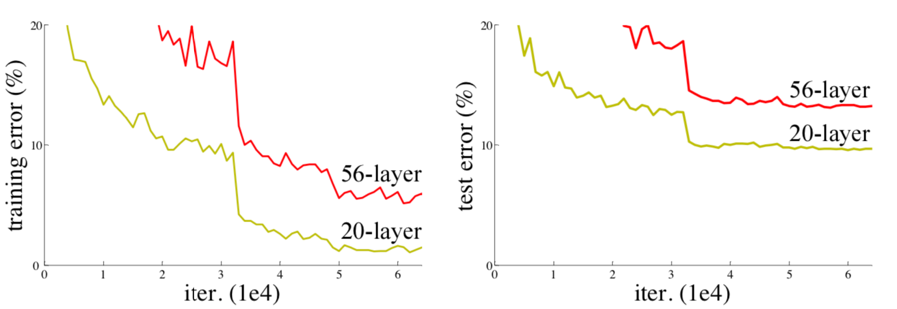
Gambar 3.10 Contoh Jaringan *Siamese*[[7]](#footnote-7)

Pada gambar 3.10 diatas *input* dan melewati dua jaringan yang sama (identik) yang memiliki nilai parameter yang sama, hasil dari jaringan ini adalah *feature vectors/embedding* dari input dan , *embedding* dari kedua *input* ini kemudian bisa dibandingkan kedekatannya dengan fungsi dimana fungsi bisa berbentuk *Euclidean distance* atau *cosine similarity* sehingga dengan *threshold* tertentu kedua *input* bisa ditentukan apakah sama (*genuine*) atau berbeda (*impostor*). Proses pemetaan dimensi gambar yang tinggi menjadi representasi berdimensi lebih rendah/kecil (*embedding/feature vectors)* juga disebut dengan *dimensionality reduction.* Ada beberapa algoritma yang bisa digunakan dalam proses *dimensionality reduction,* dalam penelitian ini penulis menggunakan algoritma *triplet loss* yang akan dijelaskan lebih rinci pada poin 3.3.1*.*

Arsitektur jaringan identik yang dapat digunakan *Siamese* bisa bermacam-macam. Merancang arsitektur jaringan sendiri adalah proses yang rumit dan membutuhkan banyak percobaan dan memakan banyak waktu. Penulis dalam melakukan penelitian ini memanfaatkan arsitektur jaringan Resnet karena kehandalannya dalam melakukan klasifikasi dan ekstraksi fitur sebuah gambar, arsitektur ini mendapatkan posisi pertama pada kompetisi *ImageNet Large Scale Visual Recognition* *Competition* (ILSVRC) 2015 dengan kesalahan (*error*) hanya 3.57% pada dataset *ImageNet*.

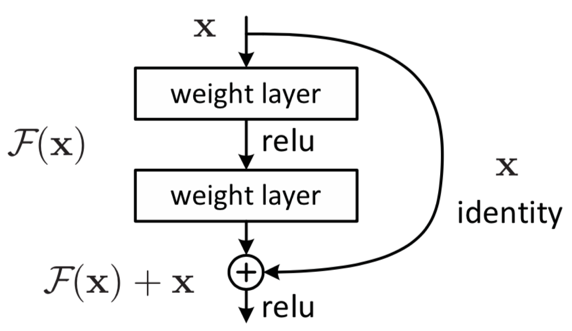
1. ***Residual Network (*ResNet)**

Arsitektur CNN yang cukup dalam telah berhasil membuat terobosan-terobosan dalam masalah klasifikasi gambar, telah banyak masalah-masalah klasifikasi gambar yang rumit memanfaatkan arsitektur model yang sangat dalam. Namun semakin dalam arsitektur sebuah jaringan semakin susah untuk dilatih dan ketika arsitektur yang sangat dalam berhasil *convergent* masalah baru seperti *degradation* muncul, dimana akurasi jaringan tersebut menurun secara cepat dan kesalahannya pun meningkat (He, Zhang, Ren, & Sun, 2015), seperti terlihat pada gambar 3.11 dibawah.



Gambar 3.11[[8]](#footnote-8) *Error* Lebih Tinggi Pada Arsitektur Yang Lebih Dalam.

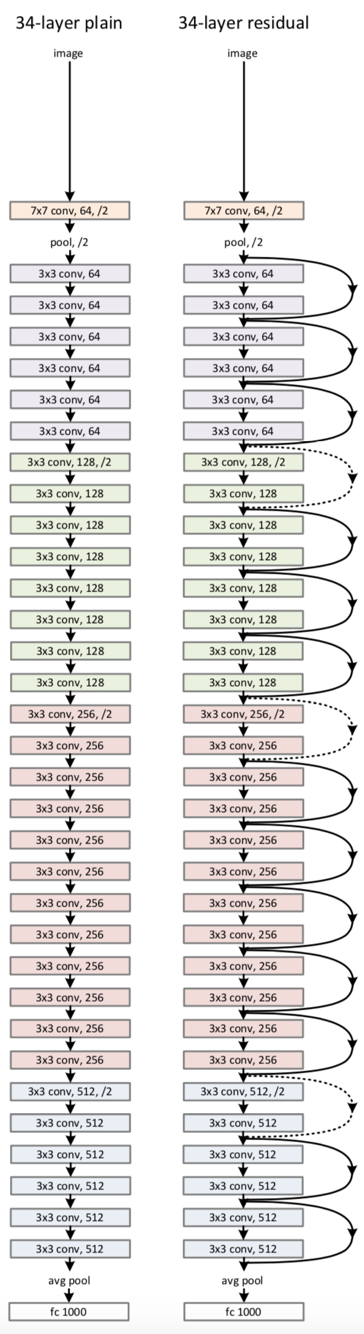
Mengatasi malasah ini kerangka arsitektur *deep residual learning* pun dibuat yang dikenal dengan ResNet (*Residual Network*). Kerangka *residual* ini memperkenalkan sambungan *shortcut* seperti gambar 3.12 dibawah.



Gambar 3.12[[9]](#footnote-9) Blok *Residual* Dengan Sambungan *Shortcut.*

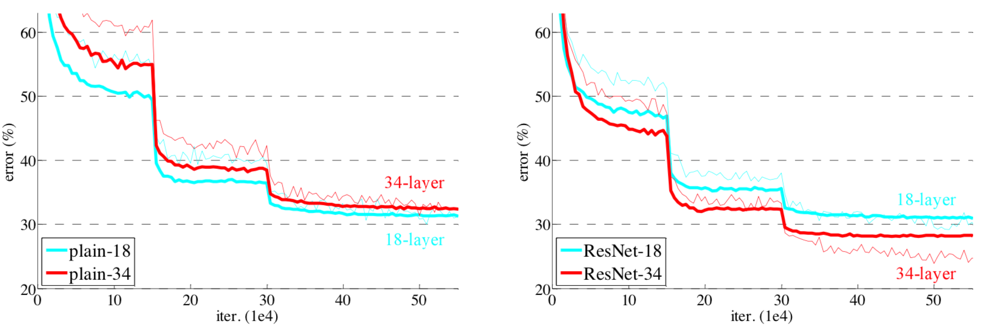
Masalah degradasi (akurasi menurun dan *error* meningkat) pada arsitektur jaringan yang sangat dalam ditemukan ketika pemetaan fungsi dari blok CNNi () bernilai 0 sehingga sangat sulit untuk menentukan parameter yang akan dipelajari pada blok CNNi+1 () karena tidak ada informasi yang dilanjutkan ke yang disebabkan oleh nilai . Disinilah peran *residual learning*,dengan melanjutkan inputan ke ujung dari blok CNNi sehingga fungsi CNNi menjadi , dengan demikian walaupun pemetaan fungsi namun informasi dari masih utuh sehingga nilai (*identity mapping*), dengan demikian fungsi tidak lagi memproses nilai 0 melainkan memproses nilai inputan yang identik dengan blok .

Proses *identity mapping* dalam blok ResNet diwujudkan dengan menambah garis *shortcut* (*skip connection*) langsung menuju ujung dari blok CNNi seperti terlihat pada gambar 3.12 diatas*.* Ini menjelaskan arsitektur ResNet mampu mengatasi masalah *degradation* pada arsitektur jaringan CNN yang sangat dalam. Contoh perbandingan arsitektur jaringan tanpa *residual block* (*plain*)dengan *residual block* dapat dilihat pada gambar 3.13 dibawah.



Gambar 3.13[[10]](#footnote-10) Arsitektur Jaringan *Plain* dan *Residual Block*

Perbandingan pembelajaran yang dihasilkan oleh jaringan *plain* dan jaringan dengan tambahan *residual block* dapat dilihat pada gambar 3.14 dibawah.

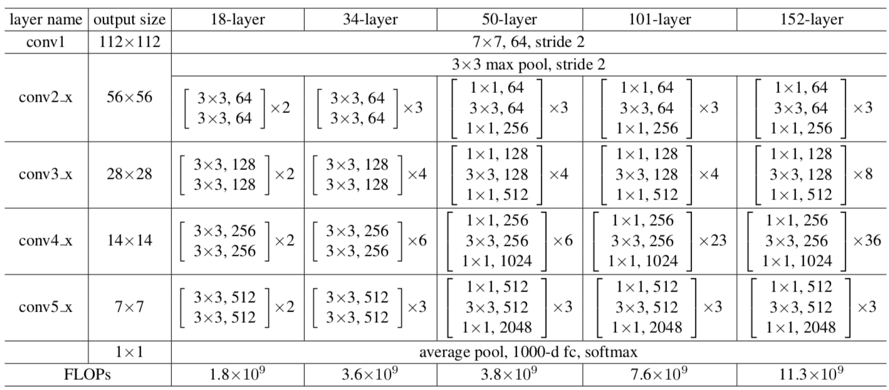


Gambar 3.14[[11]](#footnote-11) Perbandingan Kesalahan ResNet dan *Plain*

Bisa dilihat pada gambar grafik 3.14 diatas jaringan yang telah ditambahkan *residual block* dapat meraih *error* yang lebih rendah dengan arsitektur yang lebih dalam.

* + 1. ResNet-50

Adapun arsitektur jaringan ResNet ada beberapa macam seperti terlihat pada gambar 3.15 dibawah.



Gambar 3.15[[12]](#footnote-12) Beberapa Jenis Arsitektur ResNet

Seperti terlihat pada gambar 3.15 diatas ResNet memiliki 5 jenis arsitektur masing-masing mewakili jumlah layer yang dimiliki. Tentunya semakin banyak layer yang dimiliki maka bertambah juga jumlah parameter yang dimiliki. Perbandingan jumlah parameter setiap model ResNet bisa dilihat pada tabel 3.5 dibawah.

Tabel 3.5 Perbandingan Jumlah Parameter Model ResNet

|  |  |
| --- | --- |
| **Model** | **Jumlah Parameter** |
| ResNet-18 | 11.689.512 |
| ResNet-34 | 21.797.672 |
| ResNet-50 | 25.557.032 |
| ResNet-101 | 44.549.160 |
| ResNet-152 | 60.192.808 |

Dalam penelitian ini penulis menggunakan model ResNet-50 dengan alasan jumlah parameter yang cukup banyak namun tidak sampai memakan sumber daya *hardware* yang besar seperti model diatasnya (ResNet-101 dan ResNet-152), adapun model ResNet-50 yang digunakan akan dilakukan beberapa modifikasi layer sehingga jumlah parameter ResNet-50 menjadi 36.353.216, adapun modifikasi layer yang dilakukan akan dijelaskan pada poin selanjutnya.

* + 1. Konfigurasi ResNet

Model ResNet yang digunakan adalah *pretrained model* yaitu model yang sebelumnya telah dilatih dalam klasifikasi objek pada dataset ImageNet. Hasil parameter yang telah dilatih pada dataset ini kemudian dimuat lalu digunakan untuk melatih dataset baru yang digunakan pada penelitian ini.

Adapun model ResNet yang telah dilatih (*pretrained*) memiliki beberapa ketentuan yang harus diikuti sebelum digunakan yaitu agar masukan gambar (*input*) harus memiliki 3 kanal warna dengan ukuran 224x224 dan setiap gambar masukan harus mengikuti data normalisasi yang sama dengan yang digunakan pada saat melatih dataset ImageNet. Tabel data normalisasi bisa dilihat pada tabel 3.1 dan 3.2 (Pytorch, 2018).

* + 1. Modifikasi Layer ResNet-50

Beberapa modifikasi arsitektur ResNet-50 yang dilakukan dijelaskan sebagai berikut:

* + - 1. Pembuangan Layer Terakhir

Arsitektur jaringan ResNet-50 tanpa layer terakhir yang dilatih menggunakan *Triplet Loss* berhasil mendapatkan akurasi yang baik (Hermans, Beyer, & Leibe, 2017). Berdasarkan hal tersebut, penelitian ini mencoba menggunakan arsitektur yang sama. Layer terakhir ResNet adalah *Average Pool.* Adapun perbandingan arsitektur sebelum dan sesudah pembuangan layer terakhir bisa dilihat pada gambar 3.16 dan gambar 3.17 dibawah.



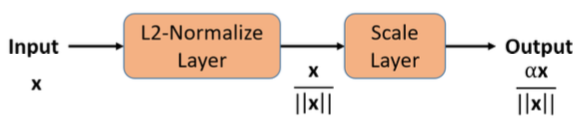
Gambar 3.16 Blok Jaringan ResNet Yang Utuh



Gambar 3.17 Blok Jaringan ResNet Tanpa Layer Terakhir

* + - 1. L2 Norm dan Skala α (*alpha*)

Penambahan L2 Norm dan skala α pada penelitian ini berdasarkan pada percobaan yang dilakukan oleh Rajeev Ranjan dkk. Rajeev menyatakan bahwa mengintegrasikan L2 Norm dan skala *alpha* pada arsitektur jaringan yang sudah ada dapat meningkatkan kinerja jaringan (Ranjan, Castillo, & Chellappa, 2017). Adapun proses yang dilakukan pada layer ini bisa dilihat pada gambar 3.18 dibawah.

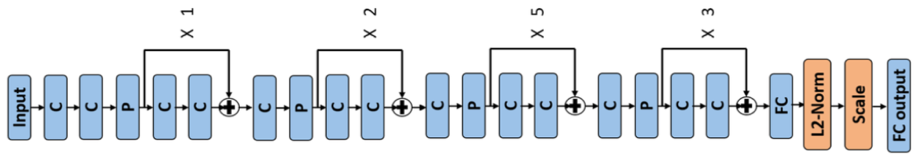


Gambar 3.18 Ilustrasi Operasi L2 Norm dan Skala α

Pada gambar 3.18 input yang diterima oleh layer L2 Norm akan diproses dengan persamaan:

Lalu hasil dari y akan dikalikan dengan nilai α, seperti dibawah:

Sesuai arsitektur yang dirancang pada paper tersebut L2 Norm dan skala α digunakan sebagai *penultimate layer* (layer sebelum output) sehingga arsitektur ResNet yang digunakan bisa divisualisasikan seperti gambar 3.19 dibawah.



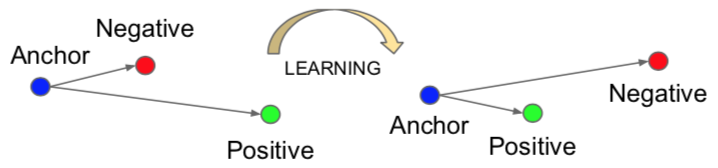
Gambar 3.19 Layer FC Setelah Penambahan L2 Norm & Skala

Adapun skala α memiliki peran penting dalam meningkatkan kinerja L2 Norm. Ada dua acara dalam menentukan nilai α, yaitu bisa dipelajari dengan melatih jaringan dengan membuat α sebagai parameter, atau dengan memberikan nilai tetep (konstanta). Dalam eksperimen Rajeev Ranjan, dengan menetapkan nilai α memberikan performa yang lebih baik. Nilai α yang digunakan pada penelitian ini adalah 10.

## Training

1. **Triplet Loss**

*Triplet Loss* adalah fungsi yang digunakan untuk mempelajari *embedding euclidean* dengan dua tujuan yaitu gambar wajah orang yang sama menghasilkan *embedding* yang berdekatan pada ruang *embedding* sedangkan gambar wajah orang yang berbeda menghasilkan *embedding* yang berjauhan pada ruang *embedding* (Schroff & Philbin, 2015)*.*



Gambar 3.20 Ilustrasi Belajar *Triplet Loss*

Adapun pada saat latihan, setiap *batch* data terdapat 3 jenis gambar yaitu *anchor, positive* dan *negative.* Gambar *anchor* adalah gambar wajah acuan, gambar *positive* adalah gambar wajah yang sama dengan *anchor* dan gambar *negative* adalah gambar wajah yang berbeda dengan *anchor*.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *Anchor* | *Positive* | *Negative* |
| Gambar 3.21 Contoh Gambar *Anchor, Positive* dan *Negative* | | |
|  | | |

Pada saat *training*, *triplet loss* akan belajar bagaimana menghasilkan *embedding* sehingga jarak antar gambar *anchor* dan *positive* menjadi lebih dekat dan jarak antar gambar *anchor* dan gambar *negative* menjadi lebih jauh seperti terlihat pada gambar 3.20 diatas. Secara matematis tujuan pembelajaran *triplet loss* bisa diekpresikan seperti persamaan dibawah:

**Dimana:**

adalah jarak antara gambar *anchor* dan gambar *positive.*

adalah nilai konstanta sebagai margin antara *positive* dan *negative.*

adalah jarak antara gambar *anchor* dan gambar *negative.*

Sehingga nilai *loss* pada setiap *minibatch* bisa dihitung dengan fungsi *triplet* *loss* dibawah:

**Dimana:**

adalah banyak data pada setiap *mini* *batch.*

adalah hasil *embedding.*

adalah input gambar *anchor.*

adalah input gambar *positive*.

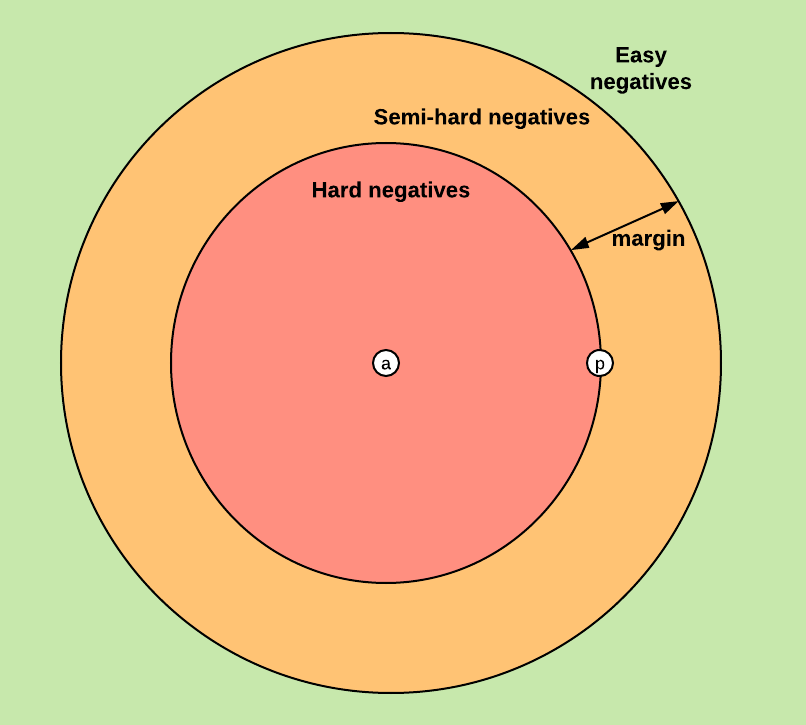
adalah input gambar *negative.*

adalah *margin.*

Menggunakan setiap pasangan *triplet* pada *minibatch* sebagai bahan latih dapat menghasilkan pembelajaran yang tidak optimal disebabkan karena ada kemungkinan pasangan *triplet* yang tidak perlu dilatih/optimasi. Sebagai contoh gambar *anchor* dan *positive* yang memang sudah dekat atau gambar *anchor* dan *negative* yang memang sudah jauh tidak perlu dimasukkan sebagai bahan latihan *triplet loss* karena tidak memiliki kontribusi yang banyak pada hasil latihan jaringan dan dapat menyebabkan *convergent* yang lama (Schroff, Kalenichenko, & Philbin, 2015). Sehingga dibutuhkan strategi dalam pemilihan pasangan *triplet* yang baik, seperti dijelaskan pada poin selanjutnya.

1. **Pemilihan Triplet**[[13]](#footnote-13)

Untuk mendapatkan hasil *embedding* yang optimal pasangan *triplet* yang digunakan untuk latihan harus baik, adapun beberapa jenis *triplet* adalah sebagai berikut:



Gambar 3.22 Ilustrasi Pembagian Jenis *Triplet*

1. Easy Triplet

*Easy triplet* adalah ketika jarak antar gambar *anchor* lebih dekat dengan gambar *positive* dibanding jarak ke gambar *negative,* *triplet* dengan jenis ini memiliki nilai *loss* 0 dan tidak perlu dioptimasi lagi.

**Dimana:**

adalah jarak antar dua gambar.

adalah *embedding* dan adalah nilai *margin*.

1. Hard Triplet

*Hard triplet* adalah ketika jarak antar gambar *anchor* lebih dekat ke gambar *negative* daripada ke gambar *positive.*

1. Semi Hard Triplet

*Semi hard triplet* adalah ketika jarak antar gambar *anchor* dan *negative* masih didalam radius jarak *anchor positive + margin*.

Adapun ketiga jenis *triplet* diatas bergantung pada posisi gambar *negative*, sehingga ketiga jenis triplet diatas bisa diberi istilah *easy negative, hard negative* dan *semi hard negative* dan bisa diilustrasikan dengan bidang lingkaran dimana jarak antar *embedding* gambar menjadi radius seperti terlihat pada gambar 3.22 diatas.

Pada penelitian ini jenis *triplet* yang digunakan untuk *training* adalah *hard* *negative triplet* dan menggunakan optimasi Adam. Contoh gambar yang termasuk *easy* dan *hard* *negative* bisa dilihat pada table 3.6 dibawah.

Tabel 3.6 Contoh Jenis Triplet

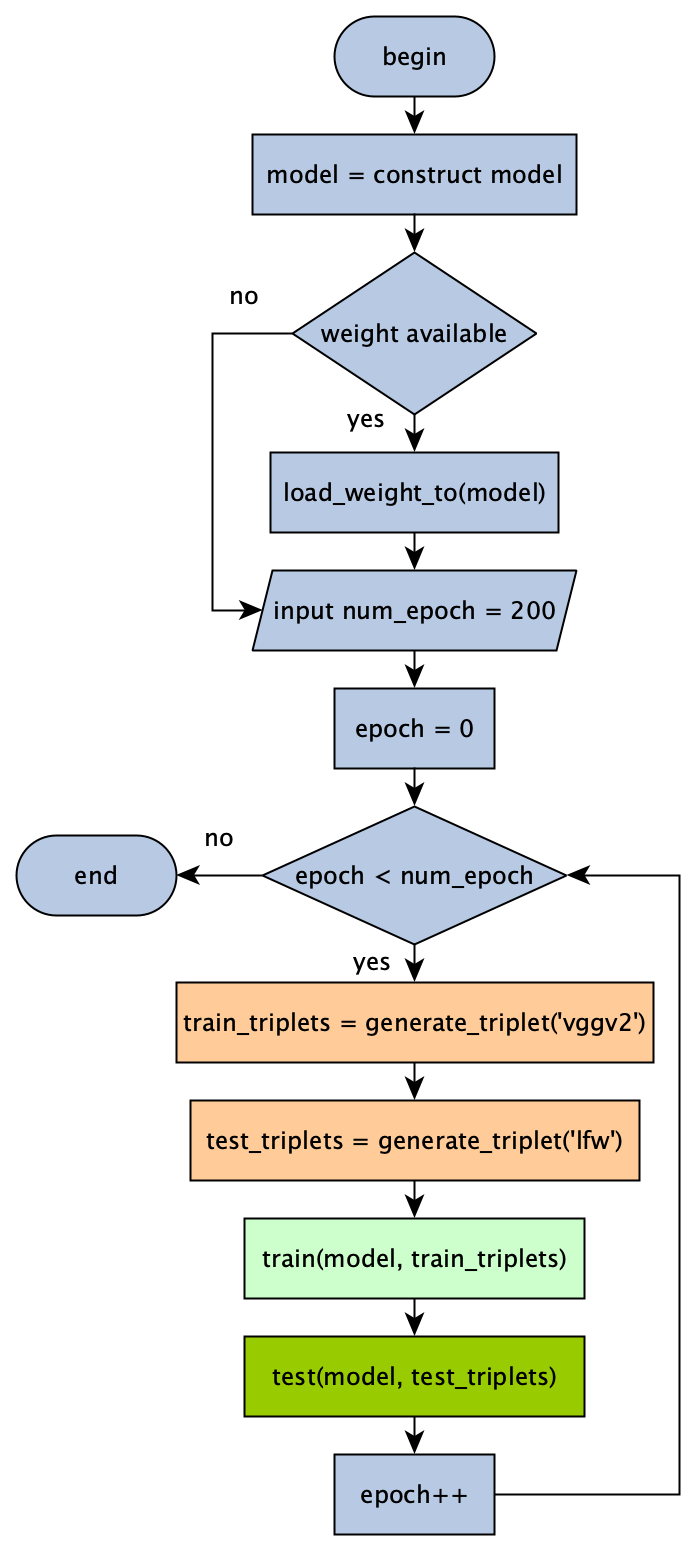
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Kategori |
|  |  |  | 10,5 | 10,9 | Ea*sy negative* |
|  |  |  | 11,26 | 9,01 | Har*d negative* |

1. **Proses Training dan Testing**

Setelah menjelaskan tahap-tahap pengumpulan dataset, persiapan dataset dan pembagian jenis-jenis triplet, pada sub bab ini akan dijelaskan tahap-tahap proses training dan testing sebagai berikut.

1. Gambaran Umum Training dan Testing

Secara umum proses training dan testing setiap *epoch* terlihat seperti gambar 3.23 dibawah.



Gambar 3.23 Gambaran Besar Proses *Training* dan *Testing*

Pada tahap konstruksi model akan dilakukan pembuatan objek model yang akan dilatih yaitu model ResNet-50 yang telah dimodifikasi seperti digambarkan pada gambar 3.17, lalu jika sebelumnya sudah dilakukan proses training maka parameter (*weight* dan bias) yang telah latih sebelumnya akan dimuat ke model yang digunakan. Adapun rincian masukan dan keluaran setiap layer pada model tersebut akan dijelaskan melalui tabel 3.7 dibawah.

Tabel 3.7 Rincian Input dan Output Setiap Layer

|  |  |  |  |
| --- | --- | --- | --- |
| **Layer** | **Input** | **Proses** | **Output** |
| Conv1 | 1x3x224x224 | Conv2D | 1x64x112x112 |
| BN1 | 1x64x112x112 | BatchNorm | 1x64x112x112 |
| ReLU | 1x64x112x112 | ReLU | 1x64x112x112 |
| MaxPool | 1x64x112x112 | MaxPool | 1x64x56x56 |
| Layer1 | 1x64x56x56 | Bottleneck1 | 1x256x56x56 |
| 1x256x56x56 | Bottleneck2 | 1x256x56x56 |
| 1x256x56x56 | Bottleneck2 | 1x256x56x56 |
| Layer2 | 1x256x56x56 | Bottleneck3 | 1x512x28x28 |
| 1x512x28x28 | Bottleneck4 | 1x512x28x28 |
| 1x512x28x28 | Bottleneck4 | 1x512x28x28 |
| 1x512x28x28 | Bottleneck4 | 1x512x28x28 |
| Layer3 | 1x512x28x28 | Bottleneck5 | 1x1024x14x14 |
| 1x1024x14x14 | Bottleneck6 | 1x1024x14x14 |
| 1x1024x14x14 | Bottleneck6 | 1x1024x14x14 |
| 1x1024x14x14 | Bottleneck6 | 1x1024x14x14 |
| 1x1024x14x14 | Bottleneck6 | 1x1024x14x14 |
| Layer4 | 1x1024x14x14 | Bottleneck7 | 1x2048x7x7 |
| 1x2048x7x7 | Bottleneck8 | 1x2048x7x7 |
| 1x2048x7x7 | Bottleneck8 | 1x2048x7x7 |
| FC | 1x100,352 | Linear | 1x128 |

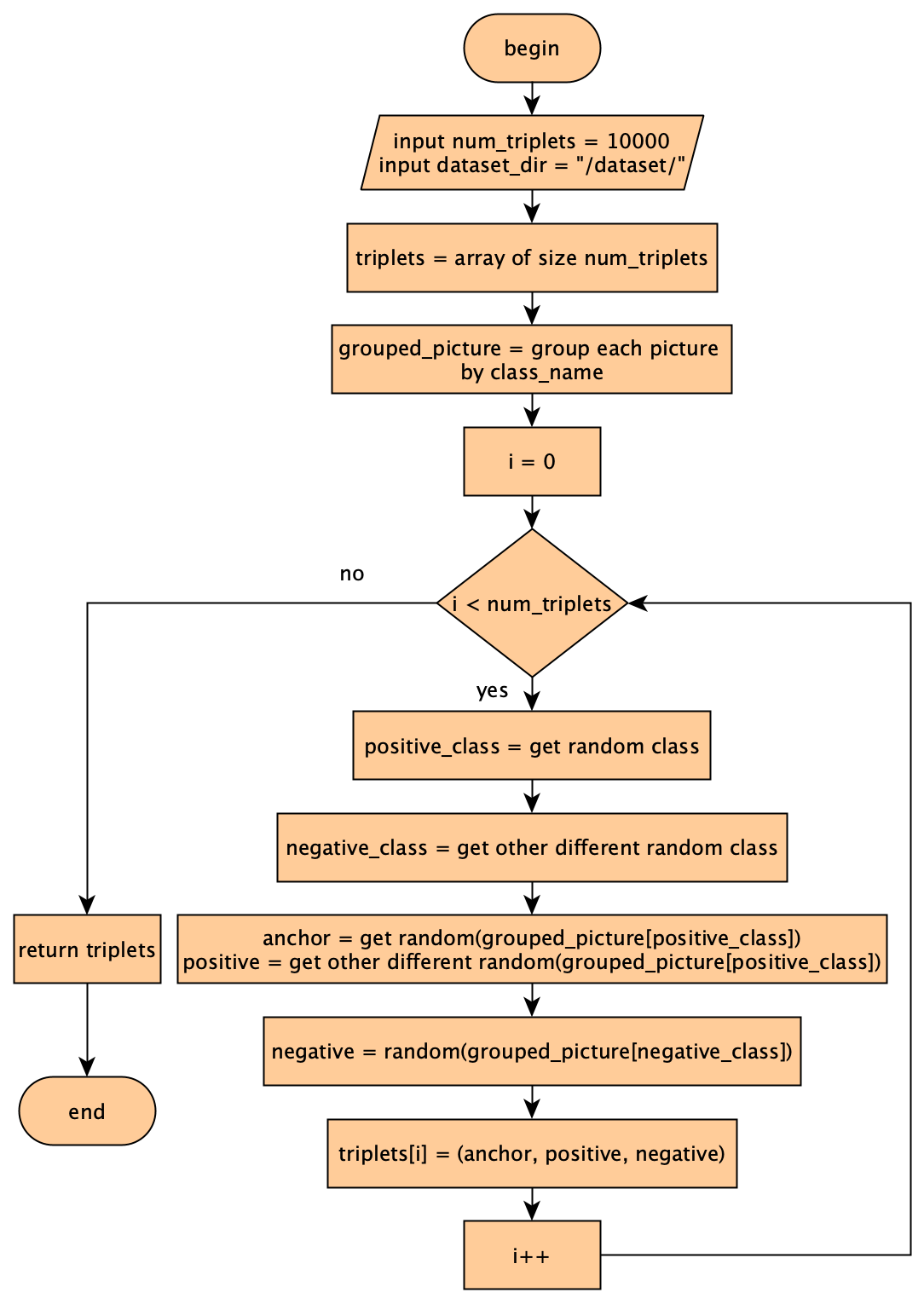
Pada tabel 3.7 diatas dirincikan masukan, proses dan keluaran setiap layer pada arsitektur ResNet-50 yang telah dimodifikasi. Sebagai contoh layer *conv1* menerima input *tensor* dengan dimensi lalu dilakukan operasi *conv2d* dan menghasilkan *tensor* dengan dimensi . Perlu diketahui angka 1 pada dimensi tensor tersebut adalah jumlah data. Sebagai contoh, jika saat training *batch size* yang digunakan adalah 64 maka jumlah data yang akan diproses adalah 64 sehingga dimensi tensor akan menjadi dan keluarannya akan menjadi . Adapun arsitektur setiap proses pada tabel 3.7 di atas dirincikan pada tabel 3.8 dibawah.

Tabel 3.8 Rincian Arsitektur Setiap Operasi Konvolusi

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Nama** | **Operasi** | **Kernel** | **Stride** | **Padding** |
| Conv2D | Conv2D | 64x7x7 | 2x2 | 3x3 |
| MaxPool | MaxPool | 3x3 | 2 | 1 |
| Bottleneck1 | Conv2D | 64x1x1 | 1x1 | - |
| BN | - | - | - |
| Conv2D | 64x3x3 | 1x2 | 1x1 |
| BN | - | - | - |
| Conv2D | 256x1x1 | 1x1 | - |
| BN | - | - | - |
| ReLU | - | - | - |
| Conv2D | 256x1x1 | 1x1 | 1x1 |
| BN | - | - | - |
| Bottleneck2 | Conv2D | 64x1x1 | 1x1 | 1x1 |
| BN | - | - | - |
| Conv2D | 64x3x3 | 1x1 | 1x1 |
| BN | - | - | - |
| Conv2D | 256x1x1 | 1x1 | 1x1 |
| BN | - | - | - |
| ReLU | - | - | - |
| Bottleneck3 | Conv2D | 128x1x1 | 1x1 | - |
| BN | - | - | - |
| Conv2D | 128x3x3 | 2x2 | 1x1 |
| BN | - | - | - |
| Conv2D | 512x1x1 | 1x1 | - |
| BN | - | - | - |
| ReLU | - | - | - |
| Conv2D | 512x1x1 | 2x2 | - |
| BN | - | - | - |
| Bottleneck4 | Conv2D | 128x1x1 | 1x1 | - |
| BN | - | - | - |
| Conv2D | 128x3x3 | 1x1 | 1x1 |
| BN | - | - | - |
| Conv2D | 512x1x1 | 1x1 | - |
| BN | - | - | - |
| ReLU | - | - | - |
| Bottleneck5 | Conv2D | 256x1x1 | 1x1 | - |
| BN | - | - | - |
| Conv2D | 256x3x3 | 2x2 | 1x1 |
| BN | - | - | - |
| Conv2D | 1024x1x1 | 1x1 | - |
| BN | - | - | - |
| ReLU | - | - | - |
| Conv2D | 1024x1x1 | 2x2 | - |
| BN | - | - | - |
| Bottleneck6 | Conv2D | 256x1x1 | 1x1 | - |
| BN | - | - | - |
| Conv2D | 256x3x3 | 1x1 | 1x1 |
| BN | - | - | - |
| Conv2D | 1024x1x1 | 1x1 | - |
| ReLU | - | - | - |
| Bottleneck7 | Conv2D | 512x1x1 | 1x1 | - |
| BN | - | - | - |
| Conv2D | 512x3x3 | 2x2 | 1x1 |
| BN | - | - | - |
| Conv2D | 2048x1x1 | 1x1 | - |
| BN | - | - | - |
| ReLU | - | - | - |
| Conv2D | 2048x1x1 | 2x2 | - |
| BN | - | - | - |
| Bottleneck8 | Conv2D | 512x1x1 | 1x1 | - |
| BN | - | - | - |
| Conv2D | 512x3x3 | 1x1 | 1x1 |
| BN | - | - | - |
| Conv2D | 2048x1x1 | 1x1 | - |
| BN | - | - | - |
| ReLU | - | - | - |

1. *Generate Random Triplet*

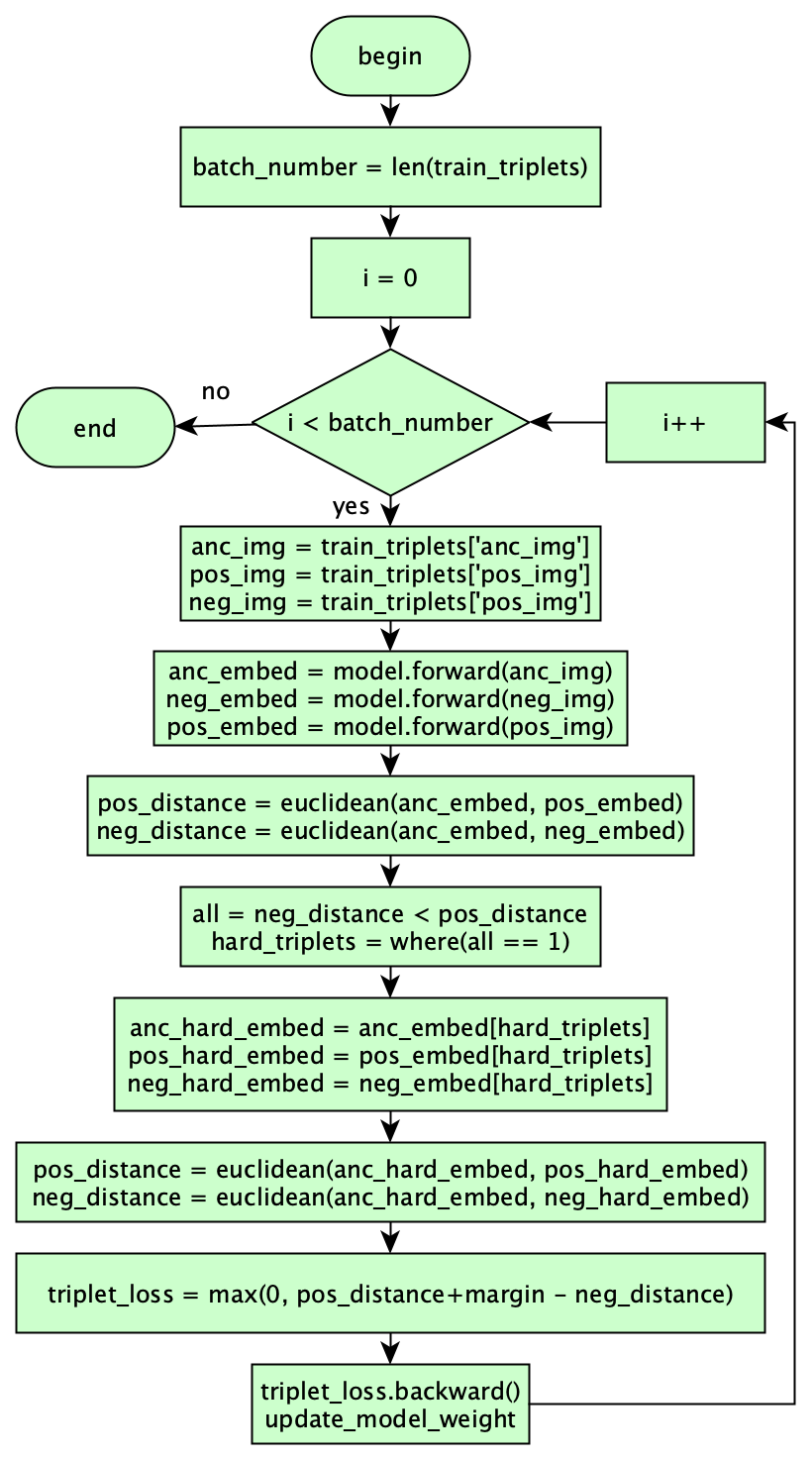
Pada tabel 3.7 dan 3.8 telah dijelaskan arsitektur CNN pada ResNet-50 yang telah dimodifikasi, setelah model dimuat lalu proses selanjutnya adalah melakukan perulangan sejumlah epoch yang diinginkan sebagai contoh 200, lalu setelah itu buat data triplet secara acak seperti dijelaskan pada gambar 3.24 dibawah.



Gambar 3.24 Flowchart Pembuatan Triplet Acak

Pada tahap awal sesuaikan jumlah data *triplet* yang diinginkan dan sesuaikan lokasi folder dataset *training/testing*. Setelah itu buat sebuah *variabel* *array* dengan ukuran sejumlah dataset yang diinginkan. Setelah itu kelompokkan setiap gambar sesuai nama folder lalu lakukan perulangan sejumlah data *triplet* yang diinginkan. Ambil gambar *anchor*, *positive* dan *negative* secara acak dan tampung pada variabel *triplets* begitu terus sejumlah triplet yang diinginkan. Setelah data triplet dibuat, data akan melalui tahap *preprocessing* seperti dijelaskan pada sub bab 3.1.4 lalu dibagi menjadi beberapa bagian (*batch size*). Pembuatan data triplet akan dilakukan pada setiap *epoch* untuk memastikan model belajar pada data yang berbeda.

1. Proses Training

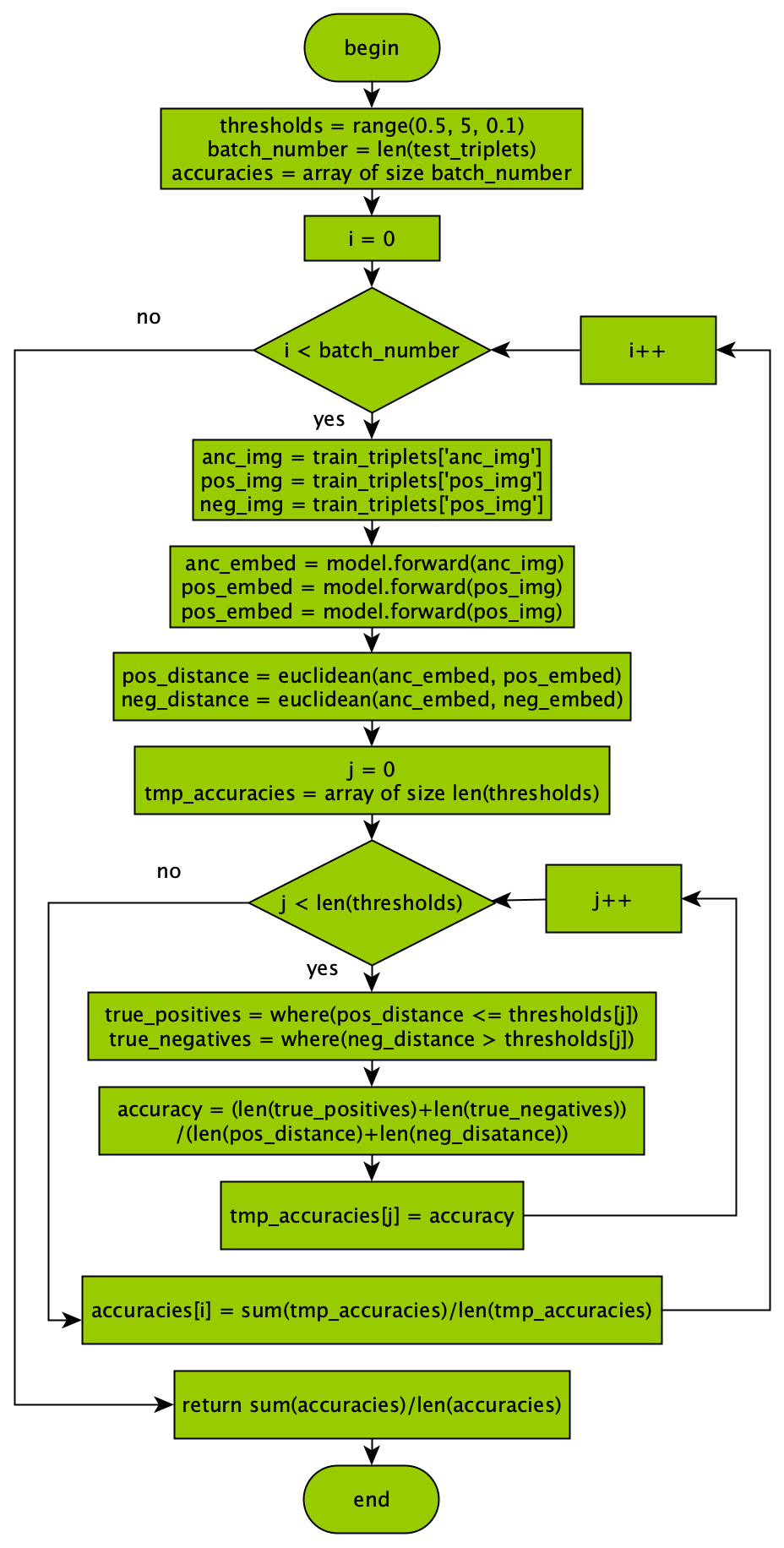


Gambar 3.25 Flowchart *Training*

Setelah proses pembuatan data *triplet* selesai proses selanjutnya adalah melakukan *training*. Alur proses training bisa dilihat pada flowchart gambar 3.25 diatas. Jumlah iterasi pada saat training adalah sesuai nilai *batch\_number* yaitu banyaknya bagian data setelah dibagi sesuai *batch\_size.*  Proses pertama pada iterasi adalah pengambilan data *anchor, positive* dan *negative* dimana setiap data adalah tensor dengan dimensi *batch\_size* x 3 x 224 x 224. Setelah itu data akan diteruskan ke model dengan operasi *forward* yaitu data akan melalui layer-layer yang telah dijelaskan pada tabel 3.7. Hasil dari operasi *forward* ini adalah *vector embedding* dari gambar *anchor*, *positive* dan *negative* lalu dilakukan penghitungan jarak antar gambar *anchor* dan *positive* dan jarak antara gambar *anchor* dan *negative*. Setelah itu akan dilakukan operasi logika untuk mencari jarak gambar *negative* yang lebih dekat ke *anchor* (*hard negative*), *index* dari gambar-gambar *hard negatives* akan disimpan di variabel *hard\_triplets* yang akan digunakan untuk mengambil vektor embedding dari gambar *anchor, positive* dan *negative*. Setelah mendapatkan gambar *anchor*, *positive* dan *negative* sesuai index *hard\_triplets* lalu dilakuan kembali penghitungan jarak dari gambar *hard­\_triplets.* Jarak yang didapat pada proses ini yang akan digunakan sebagai acuan untuk menghitung *triplet* *loss* sehingga jarak antar gambar *anchor* ke *positive* bisa diminimalkan. Setelah penghitungan loss dilakukan lalu model melakukan operasi *backward* (*backpropagation*) kemudian melakukan pembaruan parameter sesuai dengan nilai *gradient* yang didapat pada proses *backpropagation.*

1. Proses Testing

Setelah proses *training* selesai proses berikutnya adalah *testing.* Adapun alur proses *testing* bisa dilihat pada gambar 3.26 *flowchart* dibawah.



Gambar 3.26 Flowchart Proses Testing

Proses *testing* dimulai dengan mendefinisikan daftar *threshold* dan variabel *accuracies* yang akan menampung nilai akurasi pada setiap iterasi, daftar nilai *threshold* yang digunakan dimulai dari 0,5 sampai 5,0 dengan perbedaan 0,1 setiap nilainya. Pada setiap iterasi data *triplets,* proses yang akan dilakukan adalah melakukan *forward* data gambar *anchor, positive* dan *negative* melalui layer-layer pada model untuk mendapatkan *vector* *embedding.* Adapun model yang digunakan pada proses forward ini adalah model dengan parameter yang telah diperbarui pada proses *training* sehingga bisa diketahui seberapa baik parameter yang telah dipelajari pada saat *training.* Setelah mendapatkan *vector embedding* dari gambar *anchor*, *positive* dan *negative* lalu proses berikutnya adalah menghitung jarak antar gambar *anchor* ke *positive* dan *negative*. Setelah mendapatkan jarak tersebut lalu dihitung berapa banyak embedding *true positive* dan *true negative* pada setiap nilai *threshold* yang telah didefinisikan. Nilai *true positive* bisa didapat dari jumlah jarak *anchor* dan *positive* yang lebih kecil atau sama dengan nilai *threshold*, dan nilai *true negative* bisa didapat dari jumlah jarak *anchor* dan *negative* yang lebih besar dari nilai *threshold*. Jumlah nilai *true positive* dan *true negative* jika dibagi dengan jumlah data pada *mini batch* akan menjadi nilai akurasi pada batch tersebut, lalu akurasi keseluruhan diambil dari rata-rata akurasi setiap *mini* *batch* yaitu jumlah semua nilai rata-rata akurasi pada setiap *mini batch* dibagi dengan jumlah batch (*batch number*). Proses *training* dan *testing* akan terus dilakukan sejumlah *epoch* yang diinginkan atau sampai model belajar dengan baik dengan diukur melalui nilai akurasi yang didapat pada proses *testing.*

1. www. robots.ox.ac.uk/~vgg/data/vgg\_face2. [↑](#footnote-ref-1)
2. Loc.cit. [↑](#footnote-ref-2)
3. www.vis-www .cs.umass.edu/lfw/sets1.html. [↑](#footnote-ref-3)
4. Zhang, K., Zhang et al. (2016). Joint Face Detection and Alignment using Multi-ta, hal 2. [↑](#footnote-ref-4)
5. Ibid, hal 3. [↑](#footnote-ref-5)
6. ﻿A. Karpathy, “Cs231n convolutional neural networks for visual recognition.,” (Standford University). [↑](#footnote-ref-6)
7. Martin et al. (2017). A convolutional siamese network for developing similarity knowledge in the SelfBACK dataset. *CEUR Workshop Proceedings*, *2028*, 85–94, hal 3. [↑](#footnote-ref-7)
8. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition, hal 1. [↑](#footnote-ref-8)
9. Ibid, hal 2. [↑](#footnote-ref-9)
10. Ibid, hal 4. [↑](#footnote-ref-10)
11. Ibid, hal 5. [↑](#footnote-ref-11)
12. Loc.cit. [↑](#footnote-ref-12)
13. Moindrot, O. (2018). Triplet Loss and Online Triplet Mining in TensorFlow | Olivier Moindrot blog. Retrieved June 28, 2019, from https://omoindrot.github.io/triplet-loss#triplet-loss-and-triplet-mining [↑](#footnote-ref-13)