# BAB II TINJAUAN PUSTAKA

.

## Tinjauan Teoritis

### *Convolutional Neural Network* (CNN)

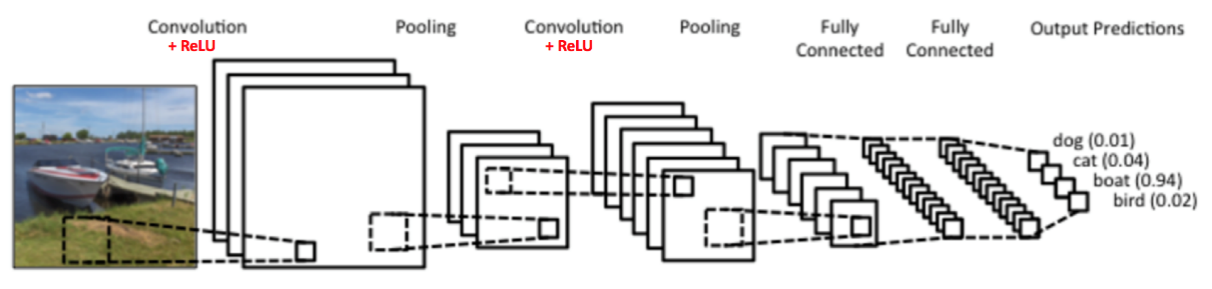
*Convolutional Neural Network* (CNN) merupakan pengembangan dan variasi dari *Multilayer Perceptron* (MLP) yang terinspirasi dari jaringan syaraf manusia, Pada tahun 2012, penelitian tentang CNN dapat melakukan pengenalan citra digital dengan akurasi yang hampir menyaingi manusia pada dataset tertentu (Coates et al, 2011),

Perlu diketahui pada dasarnya bahwa, CNN berbeda dengan arsitektur jaringan syaraf tiruan pada umumnya, dimana pada CNN menggunakan layer-layer yang memiliki susunan tiga dimensi. Dimana ini memuat tinggi, lebar, dan juga ke dalaman. Lebar dan tinggi merupakan ukuran layer itu sendiri, sedangkan ke dalaman mengacu pada jumlah *layer* yang digunakan.

CNN memiliki susunan neuron 3D (lebar, tinggi dan ke dalaman) seperti yang digambarkan pada gambar 2.7 dan 2.8, tinggi dan lebar ini merupakan ukuran yang merepresentasikan resolusi dari *layer* dalam jaringan CNN, sedangkan ke dalaman disini mengacu pada jumlah tumpukan *window* dan *feature-map* hasil dari proses operasi konvolusi antara *input* citra dan setiap *window* atau *kernel* dalam satu layer konvolusi. Selanjutnya barulah CNN mengadopsi jaringan syaraf tiruan yang disebut dengan *fully connected layer.*



Gambar 2.7 Ilustrasi Arsitektur *Convolutional Neural Network (*Deshpande, 2016*)*



Gambar 2.8 Ilustrasi Arsitektur *Convolutional Neural Network II (*Karn, 2016*)*

Secara umum jenis layer pada CNN dibedakan menjadi dua, yaitu bagian ekstraksi fitur dan bagian klasifikasi.

Untuk *Layer* bagian ekstraksi fitur citra, tersusun atas beberapa layer dan setiap *layer* tersusun atas *neuron* yang terkoneksi pada daerah lokal (*local region*) *layer* sebelumnya. Bagian ini terdiri *convolution layer* dan layer jenis kedua adalah *pooling layer* atau *subsampling layer*.

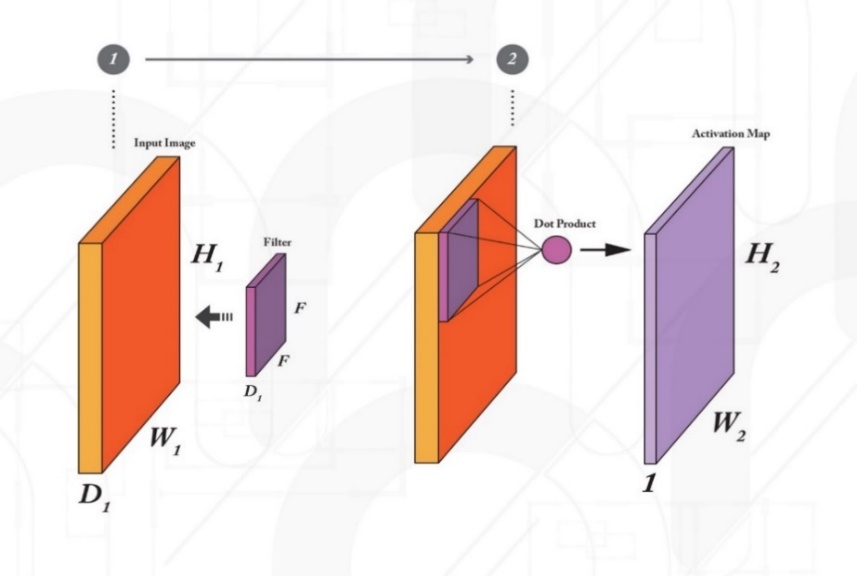
Untuk *layer* klasifikasi tersusun atas beberapa layer dan, setiap layer tersusun atas neuron yang terkoneksi secara penuh (*fully-connected*). *Fully connected Layer* inilah yang akan bertugas dalam menentukan *output* class dari setiap *input* citra yang diberikan.

Adapun penjelasan detail dari masing masing kernel untuk setiap layer dapat disimak pada penjelasan di bawah ini.

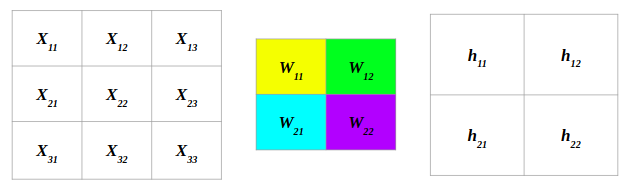
1. **Convolution Layer**

Konvolusi pada metode CNN bertujuan untuk mengekstraksi fitur dari citra suatu citra. Konvolusi bersifat dapat menjaga hubungan spasial antara setiap piksel dengan mempelajari fitur gambar melalui matrikss kecil data, dimana pada setiap *convolution* *layer*, citra masukan akan dikenakan sebanyak N buah *filter (kernel)* dengan elemen random. Nilai N ini disebut *depth* dari sebuah *convolution* *layer*. Jumlah N inilah yang akan menentukan jumlah *feature map* yang dihasilkan. Setiap elemen pada matrikss *filter* merupakan apa yang disebut bobot dalam jaringan syaraf tiruan. Ilustrasi yang menggambarkan proses konvolusi secara sederhana pada Gambar 2.9.

Gambar 2.9 merupakan representasi *layer* konvolusi, citra yang menjadi *input* akan mengalami proses konvolusi dengan *filter* yang telah dibentuk. Untuk setiap posisi pixel dalam gambar, proses konvolusi akan menghasilkan sebuah angka yang dimana ini merupakan *dot product* antara bagian dari citra dengan *filter* yang digunakan (aktif).



Gambar 2.9 *Ilustrasi Proses Pada* Layer Konvolusi (Dharmadi, 2018)



Gambar 2.10 Ilustrasi perhitungan proses konvolusi (Agarwar, 2017)

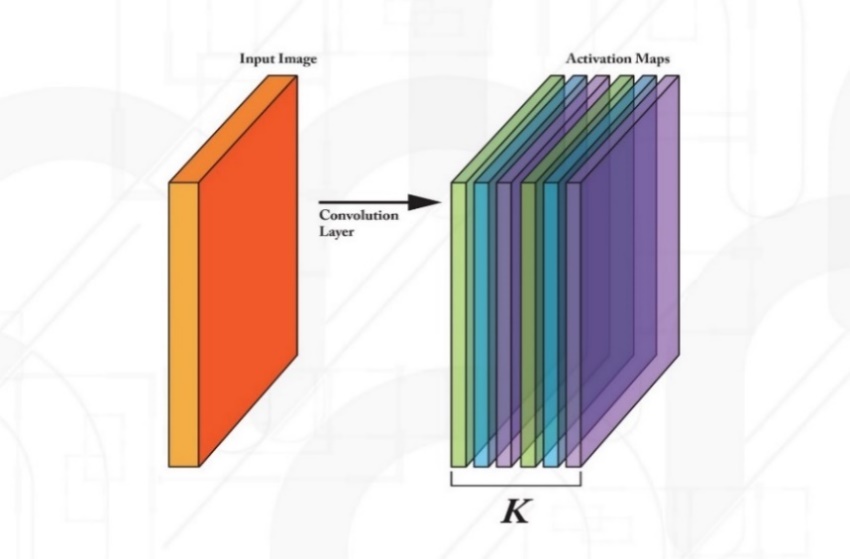
|  |  |  |
| --- | --- | --- |
|  |  | (2.2) |
|  |  | (2.3) |
|  |  | (2.4) |
|  |  | (2.5) |

Selanjut dilakukan pergeseran (*stride)* dari *kernel* atau *filter.* Setiap pergeseran akan dilakukan operasi “dot” antara data *input* dan nilai dari *filter* (*mask*). Operasi “dot” digunakan karena operasi vektor matrikss ini akan menghasilkan suatu scalar sebagai *output* dari sel yang dimana sesuai dengan dasar persitungan dari *neural network* itu sendiri. Dapat dilihat pada gambar 2.10 dan persamaan 2 hingga 5 penjabaran dari perhitungan dalam proses konvolusi yang dapat digantikan oleh operasi “dot” matrikss. *Hyperparameter* pada tahap atau layer ini dijabarkan pada tabel 2.6.

Tabel 2.6 *Hyperparameter filter* pada *Convo Layer*

|  |  |
| --- | --- |
| **Nama *Hyperparameter*** | **Keterangan** |
| Ukuran *window* | Merupakan ukuran tinggi dan lebar dari *window* yang digunakan. |
| *Stride* | Ukuran pergeseran *window* terhadap citra *input* pada proses konvolusi. |

Prose perhitungan dan konvolusi tersebut akan menghasilkan sebuah *output* atau yang biasa disebut dengan *activation map* atau *feature map* seperti yang dapat dilihat pada gambar 2.11.

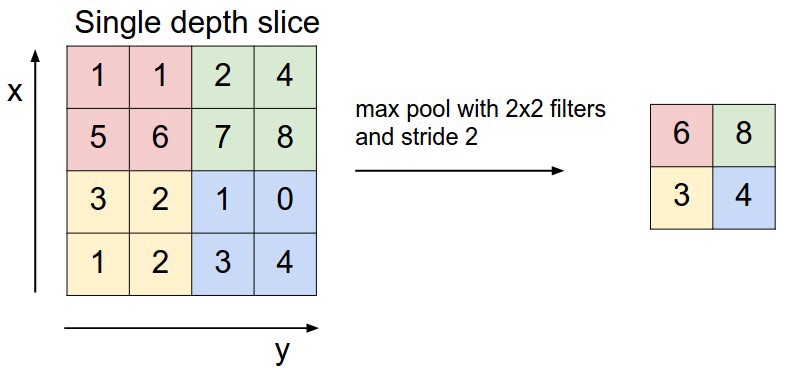


Gambar 2.11 Ilustrasi *Output* Proses Pada Layer Konvolusi (Dharmadi, 2018)

1. **Pooling Layer**

*Pooling* dapat juga disebut *subsampling* atau *down-sampling* adalah proses untuk mengurangi dimensi dari *feature map*. Adapun fungsi dari *pooling* adalah sebagai berikut (Karn, 2016).

1. Membuat representasi citra masukan (dimensi fitur) menjadi lebih kecil dan lebih mudah diatur,
2. Mengurangi jumlah *parameter* dan perhitungan pada jaringan,
3. Membuat jaringan tidak berubah terhadap sedikit transformasi, distorsi, dan translasi pada citra masukan.



Gambar 2.12 Operasi *max pooling* (Sena, 2017)

Gambar 2.12 menunjukan ilustrasi *max pooling* yang merupakan salah satu jenis *pooling*, *max pooling* tersebut memiliki *window* 2 x 2. Pada gambar 2.12 dapat dilihat bahwa citra awal berukuran 4 x 4 piksel diproses melalui *max* *pooling* dengan *window* berukuran 2 x 2 dan *stride (*pergeseran*)* 2. Hasil dari proses *max* *pooling* dapat dilihat yaitu berupa matrikss dengan ukuran 2 x 2 dimana nilai dari setiap elemen merupakan nilai terbesar dari matrikss *input* yang dilewati filer

Pada proses *backward* dalam *training*, ukuran matrikss yang mengecil akan dikembalikan, dimana dengan ketentuan nilai yang di-*backward* akan menempati posisi *max (*posisi awal pada matrikss awal*)* pada saat *Feedforward,* sedangkan posisi lain bernilai 0. Adapun beberapa *hyperparameter* dalam layer ini sama dengan *hyperparameter* pada *layer* konvolusi yang dijabarkan oleh tabel 2.6.

1. **Fully Connected Layer (*FC Layer*)**

*Fully Connected* (FC) *layer* atau *dense layer* merupakan *layer* yang pada umumnya terdapat pada jaringan syaraf tiruan tradisional. *Fully connected* disini berarti setiap *neuron* *input* terhubung dengan setiap neuron *output* (Karn, 2016). *Layer* ini menerima *input* dari hasil keluaran *layer* ekstraksi fitur dari citra yang berupa vektor ataupun *feature map*, kemudian ditransformasikan seperti *Multi Neural Networks* dengan tambahan beberapa *hidden layer*. Hasil keluaran (*output)* pada *fully connected (fc)* adalah klasifikasi berupa skoring kelas. Pada FC *layer* ini terdapat bobot-bobot yang menghubungkan *input* dengan setiap *neuron*. Pada FC *layer* terjadi proses perkalian vektor dengan matrikss yang dituliskan dalam persamaan 2.6 (Zhang, 2016).

|  |  |  |
| --- | --- | --- |
|  |  | (2.6) |

Perhitungan pada saat *feed backward* sebagai berikut:

|  |  |  |
| --- | --- | --- |
|  |  | (2.7) |
|  |  | (2.8) |

Dimana masing-masing X adalah *input*, Y adalah *output*, W adalah bobot, berturut-turut adalah gradien *error* terhadap *input, output*, dan bobot.

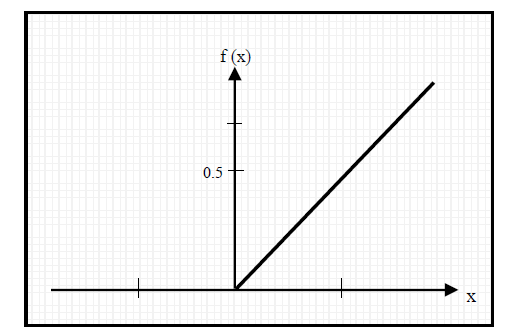
Pada FC *layer* terdapat sebuah *hyperparameter* yaitu jumlah *neuron*. Jumlah *neuron* dan panjang vektor *input* nantinyaakan menjadi penentu dari banyaknya bobot yang terdapat pada FC *layer*.

1. **Fungsi Aktivasi ReLu**

Gagasan utama dari penggunaan fungsi aktivasi yakni agar jaringan syaraf dapat bersesuaian dengan suatu fungsi non-linear, suatu jaringan syaraf tiruan setidaknya memuat dua *hidden layer* atau lebih, *hidden layer* tersebut harus menggunakan fungsi aktivasi non-linear sehingga jaringan secara keseluruhan dapat bersesuaian dengan suatu permasalahan *non-linear* (Kinsley Kinsley, 2020).

ReLU (Rectified Linear Unit) adalah salah satu jenis dari banyak fungsi aktivasi yang dapat digunakan dalam implementasi Neural Network. Dari penelitian yang telah dilakukan sebelumnya tentang *Convolutional Neural Network* sebagian besar menggunakan ReLU sebagai fungsi aktivasi. Prinsip dasar dari Operasi ReLU yaitu fungsi ini akan menerima suatu nilai dari neuron dan jika nilai tersebut berupa bilangan negatif maka fungsi ini menerjemahkan nilai tersebut ke dalam nilai 0, dan jika masukan bernilai positif maka *output* dari neuron adalah nilai aktivasi itu sendiri (Nurkhikmat, 2018).

Operasi ReLU diterapkan pada setiap piksel dan menggantikan semua nilai piksel negatif pada *feature map* menjadi nol. Jika nilai *input* dari fungsi aktivasi adalah positif, maka *output* dari *neuron* adalah nilai *input* aktivasi itu sendiri seperti yang ditampilkan pada gambar 2.13. Secara matematis operasi ReLU dapat dilihat pada persamaan 9.



Gambar 2.13 Ilustrasi Grafik Fungsi Aktivasi *ReLU* (Salsabila, 2018)

|  |  |  |
| --- | --- | --- |
|  |  | (2.9) |

Salah satu contoh dari perhitungan ReLU adalah sebagai berikut. Misalkan X adalah sebuah vektor dari suatu matrikss 1 dimensi dimana:

x = *[1, -5, 4, 3]*

maka dengan menggunakan persamaan 2.8,

*ReLU(x) = max (0, [1, -5, 4, 3])*

*ReLU(x) = [1, 0, 4, 3]*

Salah satu kelebihan ReLU dapat dilihat pada perhitungan di atas, dimana persamaan yang digunakan relatif sederhana dibandingkan dengan fungsi aktivasi lain seperti sigmoid dan tanh. Terlebih juga saat ini fungsi aktivasi ReLu adalah fungsi aktivasi yang paling sering digunakan (populer) saat ini, berkat kisah sukses pada *image recognition* dan juga *speech recognition* (Hintol et al, 2010).

Jaringan syaraf dengan fungsi aktivasi ReLU cenderung belajar beberapa kali lebih cepat daripada jaringan serupa dengan fungsi aktivasi *saturating*, seperti logistik atau tangen hiperbolik (Krizhevsky, 2012)

Fakta bahwa fungsi aktivasi ReLU tidak jenuh (*saturated*), seperti fungsi sigmoid, sifat ini terbukti berguna dalam penelitian pengenalan citra beberapa waktu belakangan (Krizhevsky, 2012).

*Neuron* logistik dikatakan jenuh ketika mencapai nilai puncaknya baik maksimum atau minimum. Dalam fungsi logistik matematika, ketika memasukkan fungsi logistik angka positif terbesar menjadi 1 dan fungsi logistik angka negatif terbesar menjadi 0. Hal ini biasa ditemukan pada fungsi aktivasi sigmoid (Krizhevsky, 2012).

|  |  |  |
| --- | --- | --- |
|  |  | (2.10) |
|  |  | (2.11) |
|  | **Saturasi:** |  |
|  |  | (2.12) |
|  |  | (2.13) |

Ketika fungsi mencapai titik maksimum ataupun minimum, dapat kita katakan bahwa fungsi tersebut sudah berada di posisi jenuh(*saturated*). Akibatnya, turunan dari fungsi logistik akan sama dengan nol pada titik jenuh. Hal ini akan menimbulkan masalah yang dikenal dengan *vanishing* gradien yang berujung pada kegagalan proses *chain rule* rambat mundur (*backpropagation*) akibat dari keberadaan gradien bernilai 0.

1. **Fungsi Aktivasi Softmax**

Pada permasalahan multiclass classification, *output* layer biasanya memiliki lebih dari 1 neuron seperti regresi logistik multinomial, analisis diskriminan linear multiclass, Naive Bayes Classifier, dan Artificial Neural Network (ANN). Untuk kasus yang berhubungan dengan multiclass classification ini, fungsi aktivasi yang biasa digunakan adalah fungsi aktivasi *Softmax*. Softmax adalah sebuah fungsi yang mengubah K-dimensi vektor 'x' yang berupa nilai sebenarnya menjadi vektor dengan bentuk yang sama namun dengan nilai dalam rentang 0-1, dan jika seluruhnya dijumlahkan maka jumlahnya 1

Dalam kasus yang menggunakan metode CNN, Softmax merupakan fungsi aktivasi yang diletakkan pada layer terakhir model, dalam hal ini maka softmax terletak pada *fully connected layer*. Softmax berfungsi untuk mengubah nilai vektor *output* menjadi nilai-nilai probabilitas dari setiap kelas label yang digunakan untuk proses klasifikasi. Jika diketahui adalah adalah *input* berbobot yang diterima oleh neuron pada softmax layer maka aktivasi untuk *neuron* ke-j dapat dilihat pada persamaan 2.14.

|  |  |  |
| --- | --- | --- |
|  |  | (2.14) |

Di mana bagian penyebut (sisi kiri) pada persamaan 2.14 merupakan total nilai masing-masing *neuron* pada *output* layer. Bisa dikatakan pada *softmax layer*, *output* merupakan distribusi probabilitas untuk setiap kelas. Penyebutnya memastikan bahwa *output* ke-i berjumlah mendekati 1. Dengan menggunakan *softmax* kita bisa menafsirkan *output* jaringan sebagai perkiraan **.**

1. **Penurunan Nilai Error (Cross Entropy Error)**

Fungsi aktivasi softmax yang diterapkan pada *fully connected layer* (fc layer) akan berpasangan dengan cross entropy. Cross entropy ini nantinya akan berfungsi dalam menghitung besar nilai loss atau error dari *output* softmax terhadap *output* yang diharapkan. Adapun *cross entropy* error dapat dihitung dengan menggunakan persamaan 2.15:

|  |  |
| --- | --- |
|  | (2.15) |

1. **Adam (*adaptive moment estimation*)**

Optimisasi Adam merupakan algoritma optimisasi yang diperkenalkan oleh D. P. Kingma and J. Lei Ba pada tahun 2015. Adam merupakan optimalisasi stokastik yang efisien yang hanya membutuhkan gradien dengan sedikit kebutuhan memori. Metode ini menghitung kecepatan pembelajaran adaptif individu untuk parameter yang berbeda dari perkiraan momen pertama dan kedua dari gradien (Kingma, 2015).

Adam saat ini adalah *optimizer* atau algoritma optimalisasi yang paling banyak digunakan dan dibangun di atas RMSProp, dengan konsep momentum dari SGD yang ditambahkan kembali. Ini berarti bahwa, bukannya menerapkan gradien terkini, algoritma ini akan menerapkan momentum seperti di algoritma optimalisasi SGD *plus* momentum, kemudian menerapkan tingkat pembelajaran adaptif per *weight* dengan *cache* seperti yang dilakukan di RMSProp (Kinsley, 2019).

|  |  |  |
| --- | --- | --- |
|  |  | (2.16) |
|  |  | (2.17) |
|  |  | (2.18) |
|  |  | (2.19) |
|  |  | (2.20) |
|  |  | (2.21) |

Dimana

= learning rate awal

= gradien pada waktu t(*step/epoch*) sepanjang

= rata rata eksponensial dari gradien sepanjang

= rata rata eksponensial dari kuadrat dari gradien sepanjang

= nilai normalisasi dari

Algoritma Adam sendiri memiliki 2 *hyperparameter* yang dapat disesuaikan berdasarkan kebutuhan dari pelatihan model. Hyperparameter tersebut adalah . Secara default masing masing parameter bernilai sebagai berikut (Kinsley, 2019):

= 0.9

= 0.999

Parameter merupakan nilai yang sangat kecil untuk menghindari pembagian 0

= 10-8

1. **RMSProp (Root Mean Square Propagation)**

Root Mean Square Propagation (RMSProp) ditemukan oleh Geoffrey Hinton(2012). Algoritma ini mirip dengan algoritma gradien *descent* *with momentum*. RMSProp mencoba menyelesaikan permasalahan *learning rate* algoritma Adagrad yang berkurang secara radikal dengan menggunakan *moving average* dari gradien kuadrat, yang memanfaatkan besarnya penurunan gradien teraktual untuk normalisasi gradien. Oleh karena itu, dengan peningkatan *learning rate*, algoritma yang digunakan akan bergerak ke arah horizontal dengan langkah-langkah yang lebih besar berkumpul lebih cepat (Yaqub, 2020).

RMSProp menggunakan konsep *Exponentially Weighted Average* (EWA) dari gradien. EWA digunakan untuk menemukan rata-rata bergerak atau *moving average*. Proses ini melibatkan penyimpanan nilai parameter sebelumnya dalam *memory buffer* dan terus memperbarui *buffer* setiap kali pengamatan baru dibaca. Proses ini ditulikan ke dalam bentuk persamaan 2.22 (Ruder, 2017).

|  |  |  |
| --- | --- | --- |
|  |  | (2.22) |
|  | Dimana  = nilai peluruhan dalam rentang  = matrikss *derivative* vektor atau gradien dari *weight* maupun *bias* layer.  = *Exponentially Weighted Average* atau *moving average* pada tahap *t* atau perbaikan terkini dari gradien *bias* maupun *weight* dari *layer*.  = tahap terkini |  |

Nilai yang disarankan dari β pada persamaan 2.22 adalah 0,9 yang sudah diamati secara eksperimental. Jika kita β kurang dari 0,9 maka fluktuasi nilai vdw dan vdb akan sangat tinggi. Di sisi lain, menjaga nilai β lebih dari 0,9 tidak akan memberikan nilai rata-rata yang tepat (Ruder, 2017). Persamaan 2.23 merupakan persamaan akhir dari algoritma optimalisasi.

|  |  |  |
| --- | --- | --- |
|  |  | (2.23) |
|  | = *bias* maupun *weight* dari layer.  = *damping factor* yang bernilai sangat kecil biasanya bernilai 10-8 untuk menghindari pembagian 0.  = *learning rate,* nilai alfa atau *learning rate* |  |

Untuk besar *learning rate* dalam implementasi algoritma RMSprop yang disarankan adalah 0,001. Menjaga alfa kurang dari 0,001 akan membuat proses pembelajaran lebih lambat, sedangkan menjaganya lebih dari 0,001 akan menghasilkan peluang *overshooting minima* (Ruder, 2017).

1. ***Backpropagation***

*Backpropagation* menghitung bagaimana mengubah setiap kekuatan sinaps sedikit demi sedikit akan mengubah *error* dari jaringan, proses ini menggunakan *chain rule* atau aturan rantai dari kalkulus. Selain itu, algoritma backprop mengerjakan proses komputasi untuk semua kekuatan sinaps pada saat yang bersamaan dan hanya membutuhkan jumlah komputasi yang sama seperti yang diperlukan oleh *forward propagation* untuk melewati jaringan. Kunci utamanya adalah menerapkan aturan rantai kalkulus menggunakan komputasi rekursif dari '*error signals*' (Lilicrap, 2020).

*Back-propagation* disebut seperti ini karena untuk menghitung turunan, kita menggunakan aturan rantai yang dimana dimulai dari lapisan terakhir (yang merupakan yang terhubung langsung ke *loss function* (persamaan 2.15), karena dalam model ini, ia secara langsung terhubung pada fungsi prediksi (*softmax* / persamaan 2.14) hingga ke lapisan pertama, atau lapisan penerima *input* citra. Pergerakan ini dapat dikatakan “bergerak dari belakang ke depan".

Dalam hierarkis jaringan syaraf multilayer, *error signals* untuk *neurons* dalam satu lapisan dihitung dari sinyal kesalahan pada lapisan di atas. Dengan demikian, komputasi *error* dimulai di lapisan akhir dan mengalir ke belakang, proses ini mengarah pada konsep gagasan nilai *error* '*backpropagating*' atau merambat mundur melalui jaringan. Setelah *error signals* dihitung untuk setiap neuron, *error* *output* dari layer akhir atau layer teratas dapat dikurangi dengan mengubah kenaikan bobot setiap *neuron* sehingga mendorong aktivitas *postsynaptic*-nya ke arah yang ditentukan oleh *error signals*.

Dalam gradien *descent*, model mencoba untuk mencapai nilai minimum dari loss *function* sehubungan dengan parameter menggunakan turunannya yang dihitung dalam *backpropagation* (persamaan 2.27 dan persamaan 2.28). Dimana konsep yang diterapkan oleh persamaan 2.27 dan persamaan 2.28 akan ditumpuk atau *stack* bersamaan dengan fungsi turunan aktivasi dari *layer* aktivasi yang digunakan oleh *layer* tersebut, seperti, ReLu persamaan 2.31 dan 2.32, *softmax* persamaan2.36 sampai persamaan 2.41, dan fungsi tujuan yakni *loss function Cross Entropy Error* pada persamaan 2.35.

Aturan rantai untuk fungsi menetapkan bahwa turunan untuk fungsi bersarang seperti sebagai berikut:

|  |  |  |
| --- | --- | --- |
|  |  | (2.24) |

Fungsi dalam jaringan dapat diinterpretasikan pada persamaan 2.25 (Lilicrap, 2020):

|  |  |  |
| --- | --- | --- |
|  | Dimana  merupakan *output* layer  merupakan *input* layer  merupakan bobot dari layer  merupakan bias dari layer  merupakan fungsi aktivasi | (2.25) |
|  | Dalam penelitian ini operasi untuk satu layer dituliskan oleh persamaan 2.26. |  |
|  |  | (2.26) |

Persamaan 2.26 mengandung 3 fungsi operasi bersarang meliputi perkalian *input* dan bobot, penjumlahan *input*, bobot dan bias, dan fungsi aktivasi yang dalam hal ini adalah . Oleh karenanya untuk mengetahui seberapa besar pengaruh dari suatu bobot pada *output* dari *layer* maka berdasarkan *chain rule* pertama – tama harus menghitung turunan dari sehubungan dengan parameternya yakni lalu mengalikan hasilnya dengan turunan parsial dari operasi penjumlahan sehubungan dengan operasi *input*. Dimana *input* ini berisi parameter yang dipertanyakan, lalu mengalikannya dengan turunan parsial dari operasi sehubungan dengan *input*. Seluruh proses ini dituliskan pada persamaan 2.27.

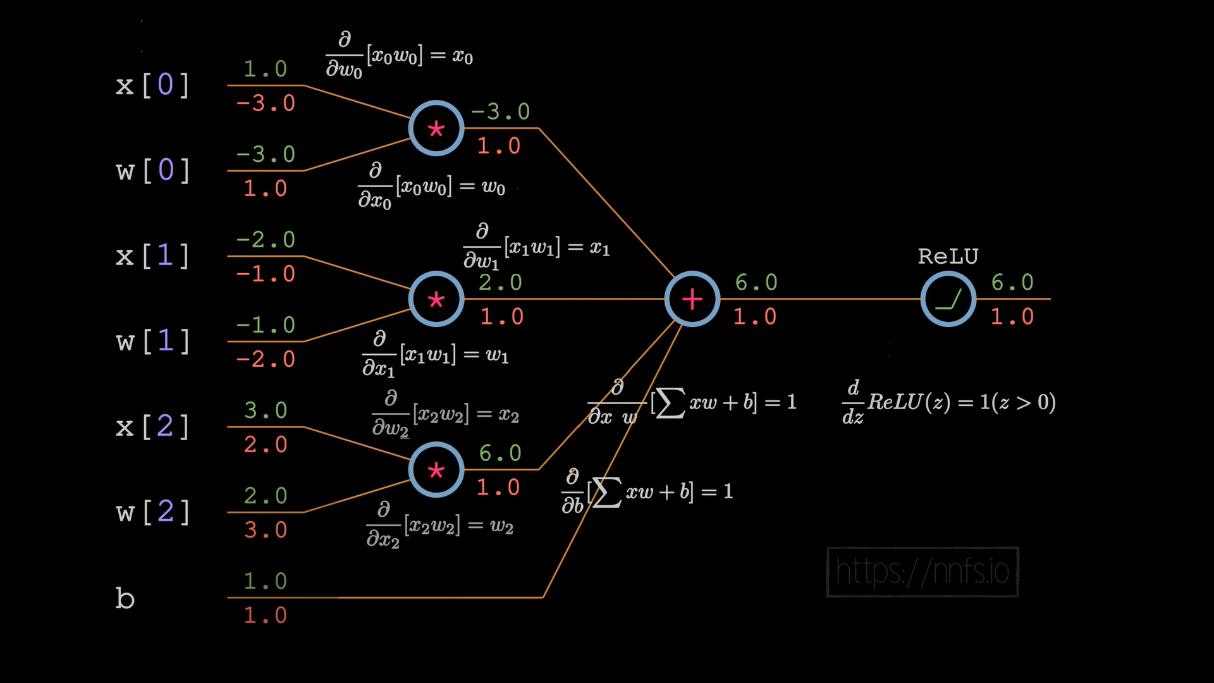
|  |  |  |
| --- | --- | --- |
|  |  | (2.27) |
|  | Maka: | (2.28) |

Persamaan 2.27 dan persamaan 2.28 menunjukkan bahwa kita harus menghitung turunan dan turunan parsial dari semua operasi *atomic* dan melipatgandakannya untuk memperoleh dampak yang dihasilkan pada *output*. Kemudian konsep ini dapat diulangi untuk menghitung semua dampak yang disebabkan oleh yang tersisa. Turunan sehubungan dengan bobot dan bias akan memberikan informasi tentang besar dampak atau pengaruh dari mereka dan akan digunakan untuk memperbarui bobot dan bias ini. Turunan sehubungan dengan *input*() nantinya akan dikirim dan digunakan untuk merantai lebih banyak lapisan dengan meneruskannya ke fungsi sebelumnya dalam rantai. Gambar 2.14 menampilkan ilustrasi proses *backpropagation* untuk 1 *layer.* Berdasarkan persamaan 2.29 turunan parsial dari operasi penjumlahan selalu bernilai 1.

|  |  |  |
| --- | --- | --- |
|  |  | (2.29) |

Maka turunan dari operasi dituliskan dalam persamaan 30, untuk setiap , dan bias dalam *layer*.

|  |  |  |
| --- | --- | --- |
|  |  | (2.30) |



Gambar 2.14 Ilustrasi Algoritma *Backpropagation* untuk 1 *layer* (Kinsley, 2019)

Gambar 2.14 merepresentasikan proses *backpropagation* untuk 1 layer. Nantinya akan ada beberapa *layer* yang berkaitan satu sama lain dalam model jaringan syaraf tiruan diikuti dengan suatu fungsi *error* atau *loss*. Oleh karenanya dalam satu *cicle* pelatihan, kita harus mengetahui seberapa besar dampak atau pengaruh dari suatu *weight* dan *bias* suatu *layer* terhadap besarnya nilai *error* atau *loss*. Kita perlu menghitung turunan dari fungsi *error* atau *loss dan* menerapkan aturan rantai dengan turunan dari semua fungsi aktivasi dan neuron di semua lapisan secara berurutan.

1. **Turunan Fungsi**

Bagian ini akan menjabarkan turunan dari setiap fungsi aktivasi yang digunakan dalam model Convolutional Neural Network yang dibangun.

***Relu***

Turunan dan turunan parsial tidak terbatas pada operasi penambahan dan perkalian, ataupun konstanta. Dalam proses *backpropagation* diperlukan turunan dari bentuk fungsi lain yang digunakan di pada tahap *Feedforward*, salah satunya adalah turunan dari fungsi max(x,y) seperti yang digunakan pada fungsi aktivasi ReLu(Kinsley Kinsley, 2020).

|  |  |
| --- | --- |
|  | (2.31) |
|  | (2.32) |

Diketahui fungsi maks mengembalikan *input* terbesar atau mengembalikan suatu konstanta. Perhatikan bahwa persamaan 2.32 mengambil satu parameter, maka persamaan 2.32 menggunakan operator d alih-alih untuk menghitung turunan non-parsial.

Berdasarkan gambar 2.14, untuk mengirimkan ‘*error signal* atau gradien’ kepada layer sebelumnya maka fungsi operasi *layer* akan diturunkan sehubungan dengan nilai x atau *input*. Untuk setiap kasus *input* x dijabarkan sebagai berikut:

Jika x lebih besar dari y, fungsi akan mengembalikan x. Diketahui bahwa turunan x sehubungan dengan x sama dengan 1, sehingga turunan dari fungsi ini sehubungan dengan x sama dengan 1.

Jika y lebih besar dari x karena fungsi akan mengembalikan y. diketahui bahwa turunan y sehubungan dengan x sama dengan 0 – dimana y diperlakukan sebagai suatu konstanta ambang batas bawah yang bernilai 0.

***Cross Entropy Error***

Turunan dari fungsi *loss* atau *error* ini sehubungan dengan (*input*) (nilai yang diprediksi pada sampel ke-i, karena proses ini membutuhkan informasi gradien sehubungan dengan nilai yang diprediksi) sama dengan vektor negatif *ground-truth* atau nilai target, dibagi oleh vektor nilai yang diprediksi oleh model (yang juga merupakan *output* vektor dari fungsi *softmax*) pada persamaan 2.33.

|  |  |  |
| --- | --- | --- |
|  | **Gradien** = | (2.33) |

Sebelumnya ingat kembali persamaan dari fungsi *Softmax:*

|  |  |  |
| --- | --- | --- |
|  |  | (2.34) |

Dimana:

nilai loss dari sampel

merupakan sampel dalam suatu set

merupakan indekslabel *output*

merupakan nilai target

merupakan predicted value

Untuk menghitung turunan dari persamaan 2.34, pertama kita harus menyelesaikan bentuk logaritma dari fungsi, yang merupakan timbal balik parameternya, dikalikan (menggunakan aturan rantai) dengan turunan parsial dari parameter ini menggunakan notasi-notasi prima (juga disebut notasi Lagrange).

|  |  |  |
| --- | --- | --- |
|  |  |  |

Maka persamaan 2.35 merupakan bentuk turunan dari persamaan 2.34 *Cross Entropy Error*

|  |  |
| --- | --- |
|  | (2.35) |

***Softmax***

Turunan dari persamaan softmax berhubungan dengan *input*nya dituliskan ke dalam bentuk persamaan 2.36.

|  |  |  |
| --- | --- | --- |
|  |  | (2.36) |

Dimana menyatakan *output* softmax ke-j dari sampel ke-i, z merupakan *input* array yang memuat list dari vector *input*(vector *output* dari layer sebelumnya), merupakan softmax *input* ke-j dari sampel ke-i, L merupakan banyaknya *input*s, merupakan softmax *input* ke-k dari sampel ke-i. untuk menyelesaikan turunan dari suatu persamaan berbentuk pembagian, dibutuhkan aturan turunan pembagian sehingga dihasilkan bentuk turunan yang harus diselesaikan pada persamaan 2.37

Maka:

|  |  |  |
| --- | --- | --- |
|  |  | (2.37) |

Selanjutnya menyelesaikan turunan yang berada pada sisi kanan *numerator* (operator pengurangan). Untuk menyelesaikan turunan yang mengandung konstanta e pada sisi kanan dari persamaan 2.37 maka digunakanlah sifat turunan eksponensial.

Maka:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Dalam jarak l sampai L, hanya akan muncul k sebanyak sekali tergantung indeks yang memuat sehingga lainnya dalam sampel akan dianggap sebagai suatu konstanta yang dimana turunan dari konstanta adalah 0.

|  |  |  |
| --- | --- | --- |
|  | Maka: |  |

Untuk menyelesaikan turunan dari persamaan yang berada disebelah kiri *numerator* sedikit berbeda dari yang sebelumnya karena tidak terdapat operator penjumlahan sehingga turunannya menjadi 0 jika j ≠ k dan jika j = k.

Untuk j = k jika bentuknya disederhanakan menurut akan terbentuk suatu fungsi softmax dengan *input* yang berbeda seperti ditunjukan oleh persamaan 2.38.

|  |  |  |
| --- | --- | --- |
|  |  | (2.38) |

Maka persamaan 42 akan disederhanakan lagi menjadi persamaan 2.39

|  |  |  |
| --- | --- | --- |
|  |  | (2.39) |

Untuk kasus dimanaj ≠ k jika dijabarkan ke dalam bentuk perkalian seperti yang ditunjukan pada persamaan 2.40 maka akan terbentuk dua persamaan softmax dengan *input* yang berbeda.

|  |  |  |
| --- | --- | --- |
|  |  | (2.40) |

Bentuk akhir turunan dari fungsi softmax sehubungan dengan *input*nya dituliskan pada persamaan 2.41.

|  |  |  |
| --- | --- | --- |
|  |  | (2.41) |

1. ***Confusion Matrikss***

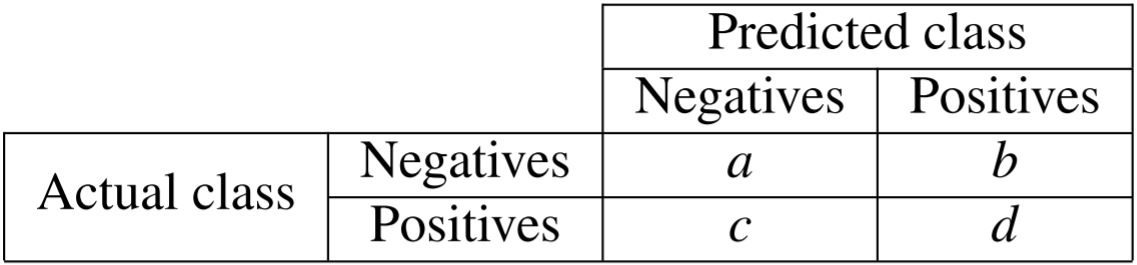
Dalam konsep proses evaluasi dari suatu model klasifikasi. Jika prediksi dari suatu model klasifikasi mengarah pada prediksi kelas yang berbeda dari kelas yang sebenarnya, maka terjadi kesalahan dalam klasifikasi. Jika setiap kesalahan itu sama pentingnya, maka total jumlah kesalahan dari suatu himpunan yang diamati (dalam hal ini hasil prediksi dari model) dapat menjadi indikator global dari kinerja suatu model klasifikasi (Novakovic, 2017).

Pendekatan ini didasarkan pada akurasi sebagai ukuran untuk mengevaluasi hasil dari model prediksi. Ukuran ini didefinisikan sebagai rasio antara jumlah prediksi benar dibagi dengan jumlah total sampel yang diklasifikasikan. Akurasi dihitung menggunakan persamaan 2.45 (Novakovic, 2017)

|  |  |  |
| --- | --- | --- |
|  |  | (2.45) |

Kelemahan utama dari akurasi yang digunakan sebagai ukuran untuk proses evaluasi yakni, mengabaikan jenis kesalahan yang terjadi dan sangat bergantung pada distribusi kelas di dalam dataset, bisa saja model hanya mengalami masalah untuk mengenali satu kelas tertentu).

Sering kali ada beberapa masalah yang menganjurkan bahwa penting untuk membedakan jenis kesalahan yang terjadi dalam model prediksi seperti membedakan kelas bunga jepun mana yang sulit untuk diidentifikasi oleh model.



Gambar 2.15 Confusion Matrikss 2 kelas

Pada gambar 2.16 dijabarkan pemetaan hasil klasifikasi untuk 2 kelas, dimana a adalah jumlah prediksi *true* negatif(model mengenali kelas negatif dan memang benar), b adalah jumlah predikti *false* positif(model mengenali kelas positif namun prediksi ini salah), c adalah jumlah prediksi *false* negatif(model mengenali kelas negatif namun prediksi ini salah), dan d adalah jumlah prediksi *true* positif(model mengenali kelas positif dan memang benar).

Selanjutnya akan dihitung nilai *presision* dan *recall* dari model dimana, presisi dan recall secara luas digunakan untuk ukuran performa suatu model klasifikasi.

*Precision* menunjukan seberapa akurat model dalam memprediksi nilai positif, atau dengan kata lain mengukur keakuratan hasil positif yang diprediksi, atau ukuran yang dapat menjadi indikator berapa sampel prediksi dari model yang merupakan prediksi yang benar. *Precision* dihitung dengan menggunakan persamaan 2.46 (Kulkarni, 2017).

|  |  |  |
| --- | --- | --- |
|  |  | (2.46) |

*Recall* berguna untuk mengukur kekuatan model dalam memprediksi hasil positif dan dikenal sebagai sensitivitas model terhadap kelas tersebut, atau untuk mengetahui sampel benar yang bisa diprediksi dengan benar oleh model yang terbentuk. *Recall* dihitung menggunakan persamaan 2.47 (Kulkarni, 2017).

|  |  |  |
| --- | --- | --- |
|  |  | (2.47) |

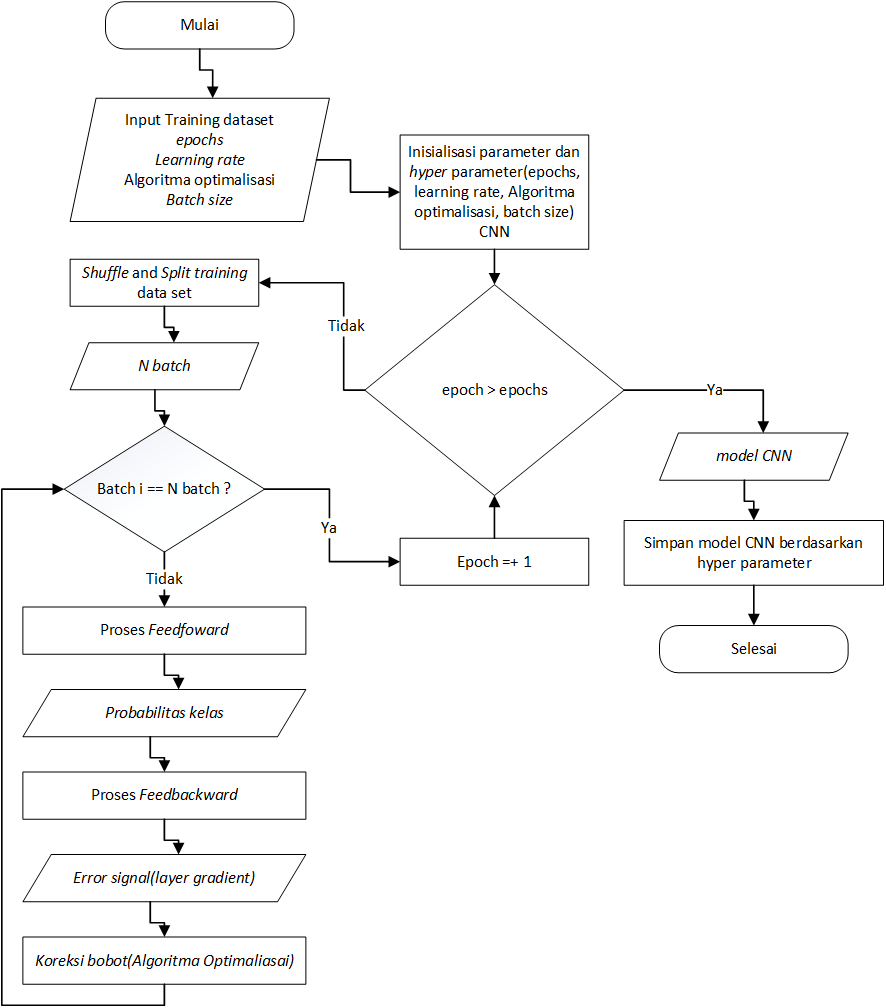
F1 atau F-measure menggunakan *precision score* dan *recall score* dari suatu model prediksi dimana ukuran ini digunakan secara luas dalam suatu kasus klasifikasi. F-measure dihitung dengan menggunakan rata rata harmonis tertimbang antara precision dan recall. Sehingga dapat dikatakan bahwa *F1-score* dapat mewakili ukuran dari *precision* dan *recall* suatu model prediksi. F1-measure dihitung menggunakan persamaan 2.48 (Kulkarni, 2017).

|  |  |  |
| --- | --- | --- |
|  |  | (2.48) |

# BAB III ANALISIS DAN PERANCANGAN SISTEM

### Tahap Pelatihan

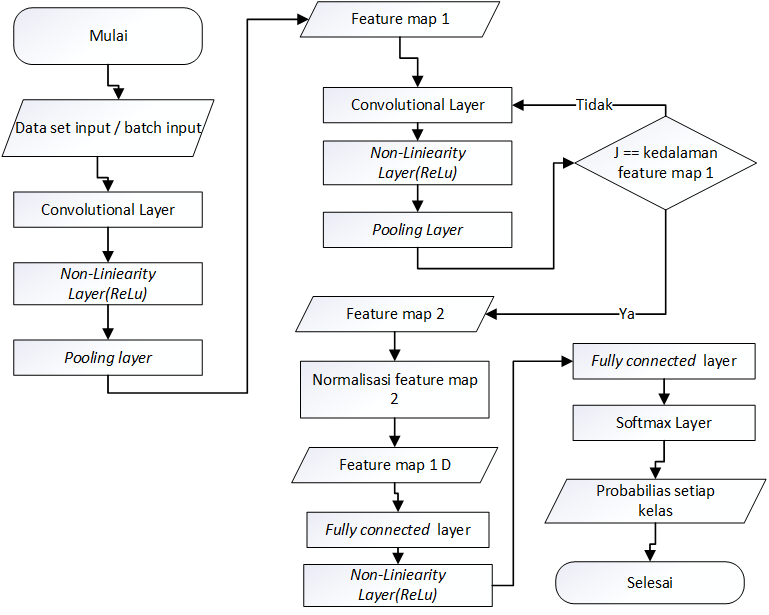
Tahap pelatihan CNN adalah tahapan yang bertujuan untuk membangun sebuah model dari metode *Convolutional Neural Network* (CNN) dan menyesuaikan model agar dapat mengenali dataset dari kasus yang diangkat. Model yang dibangun merupakan model spesifik untuk setiap algoritma optimalisasi dan parameter yang pilih dalam tahap pelatihan.



Gambar 3.5 Diagram Alir atau *flowchart* Tahap Pelatihan.

Proses pada tahap pelatihan terbagi menjadi 2 tahapan utama yaitu proses feedforward dan proses *feed backward*. *Feedforward* merupakan arah pemrosesan data *input* hingga menghasilkan *error value* pada layer paling ujung di dalam model, sedangkan *feed backward* merupakan proses pengiriman nilai *error* atau *error signal* dan perbaikan dari model CNN itu sendiri. Dimana *error signal* akan merambat mundur (*backpropagation*) dari layer klasifikasi hingga layer *input* yang dilanjutkan dengan proses koreksi bobot oleh algoritma optimalisasi. Alur dari tahap pelatihan model CNN dapat dilihat pada gambar 3.5.

Tahap pelatihan diawali dengan inisialisasi bobot secara random kemudian proses pelatihan yang mana pada setiap *epoch*, sejumlah *input* citra akan diproses secara feedforward pada model CNN. *Feedforward* berarti data *input* diproses dari *layer* paling awal hingga *layer* klasifikasi (fully connected *layer*) dan menghasilkan *output* label. *Epoch* adalah keseluruhan citra pelatihan sesuai kelompok data yang diambil secara random.



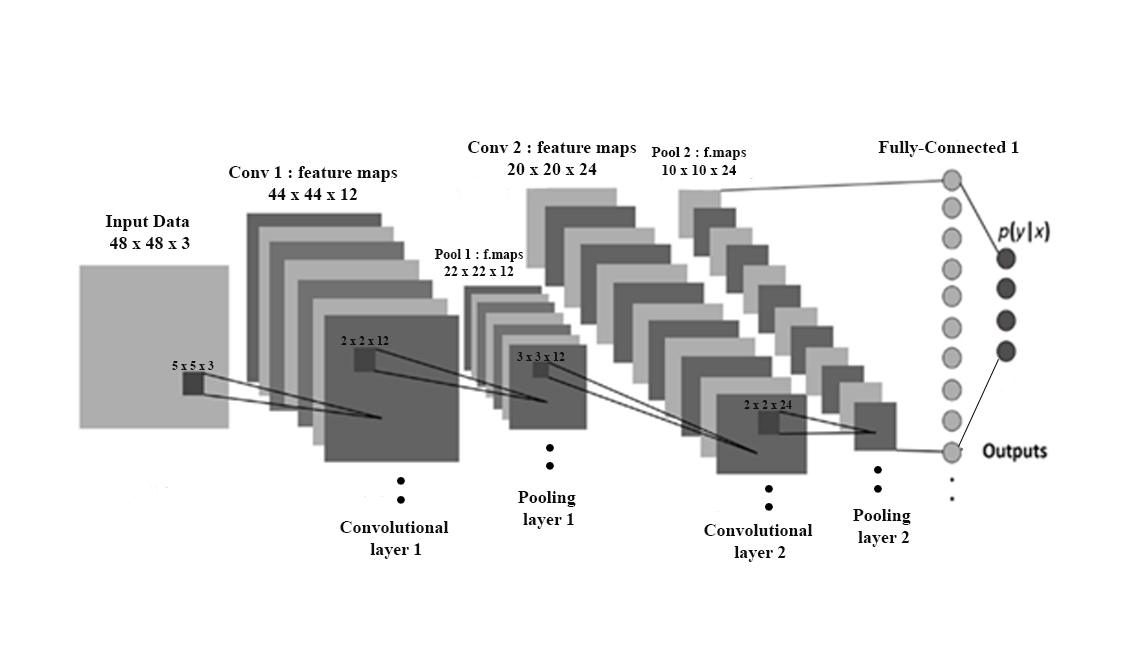
Gambar 3.6 Flowchart Tahapan *Feedforward* dari CNN

Dalam penelitian ini seluruh data training tidak akan diproses secara sekaligus dalam 1 *epoch* pelatihan, melainkan *training* data akan dibagi ke dalam beberapa *batch* atau kelompok data berdasarkan *batch size atau* ukuran tertentu. *Training* model dengan menggunakan *small batch* terbukti memberikan peningkatan kinerja generalisasi dan penggunaan memori yang jauh lebih kecil, dengan begitu processor akan menjadi lebih efisien dalam menggunakan sumber daya memori (Masters, 2018).

Dengan menerapkan pembagian *batch data,* 1 *epoch* terdapat sebanyak *n* kali proses *feedforward, back propagate*, dan koreksi yang dilakukan. Dimana *n* merupakan *n* banyak kelompok yang membagi *training* data. Proses pemecahan data latih menjadi *batch-batch* data dilakukan secara acak sehingga setiap proses pelatihan memiliki formasi data di dalam *batch* atau kelompok yang berbeda.

Gambar 3.6 menampilkan *flowchart* algoritma feedforward dalam CNN. Proses diawali dengan meng-*input*-kan citra untuk *training* hasil *normalisasi* dan *preprocessing* untuk setiap scenario ruang warna dengan ukuran 64 kali 64 px. Setelah dibangun sebuah model CNN yaitu LeNeT-5 maka citra *input* dari dataset masing masing skenario akan melalui proses pengenalan citra. Citra akan melalui proses pada setiap *layer.*

Untuk detail dari setiap *layer* yang akan digunakan pada arsitektur CNN LeNet-5 dalam penelitian ini akan diperjelas seperti pada gambar 3.7 dan tabel keterangan 3.2.



Gambar 3.7 Ilustrasi Dari Arsitektur CNN yang digunakan

Tabel 3.1 Keterangan Detail Arsitektur CNN

|  |  |  |  |
| --- | --- | --- | --- |
| No | Nama *Layer* | Jenis *Layer* | *Hyperparameter* |
| 1 | *Convo Layer* 1 | *Convolution Layer* | Jumlah *kernel*: 6  Ukuran *kernel*: 5 x 5 *Stride*: 1  *Padding*: 2 |
| 2 | ReLu 1 | Fungsi aktivasi |  |
| 3 | *Pooling* *Layer* 2 | *Pooling* *Layer* | Ukuran *window*: 2 x 2  Stride: 2 |
| 4 | *Convolution* *Layer* 3 | *Convolution* *Layer* | Jumlah *kernel*: 6  Ukuran *kernel*: 3 x 3 *Stride*: 1  *Padding*: 0 |
| 5 | ReLu 3 | Fungsi aktivasi |  |
| 6 | *Pooling* *Layer* 4 | *Pooling* *Layer* | Ukuran *window*: 2 x 2  Stride: 2 |
| 7 | FC *Layer 5* | *Fully Connected Layer* | Data *Input*: 1350  *Neuron:* 675 |
| 8 | ReLu 5 | Fungsi aktivasi |  |
| 9 | FC *Layer 6* | *Fully Connected Layer* | Data *Input*: 675  *Neuron:* 84 |
| 10 | Relu 6 | Fungsi aktivasi |  |
| 11 | *Output Layer* | *Fully Connected Layer* | Data *Input: 84*  *Neuron: 4* |
| 12 | *Softmax* | Fungsi aktivasi | *Output* label kelas |

Gambar 3.7 merupakan ilustrasi simulasi saat terjadinya pemrosesan data dalam klasifikasi CNN, dimulai dari data fitur citra yang masuk dengan ukuran citra 64 kali 64 px, kemudian akan mengalami proses ekstraksi fitur pertama pada *convolutional layer* dan *pooling layer*.

Pada convolutional *layer* pertama, fitur citra akan mengalami operasi konvolusi menggunakan persamaan 2.1 sampai 2.6 seperti yang sudah diilustrasikan pada gambar 3.7. ukuran dari *convolutional kernel*, besar pergeseran(*stride*) dan jumlah *filter* dapat dapat dilihat pada tabel 3.2.

Sebelum disimpan, nilai dari setiap proses konvolusi akan melewati *layer* aktivasi yang memuat fungsi aktivasi yaitu ReLu (persamaan 2.9). Fungsi aktivasi ini berfungsi untuk meniadakan elemen bernilai negatif pada matrikss hasil konvolusi, perhitungan dari proses aktivasi ReLu dapat dilihat pada persamaan 2.9. Ilustrasi hasil konvolusi dapat dilihat pada gambar 3.7, sedangkan visualisasi fungsi aktivasi dapat dilihat pada gambar 2.13.

Matrikss hasil konvolusi yang telah melewati *layer* aktivasi, selanjutnya akan melalui proses *max pooling* dalam *pooling layer.* Detail dari ukuran kernel dan *stride* dari *pooling layer* dapat dilihat pada tabel 3.3. ilustrasi dari proses *max pooling* dapat dilihat pada gambar 2.12. Hasil dari proses ekstraksi fitur pertama nantinya akan disimpan dalam bentuk *feature map.* Sebut saja *feature map* 1(satu) nantinya akan kembali mengalami proses ekstraksi fitur kedua.

*Feature map 1* akan melalui proses yang sama seperti pada proses ekstraksi fitur pertama, namun terdapat perbedaan dari ukuran filter pada proses kedua seperti yang telah dicantumkan pada tabel 3.3, hasil esktrasi fitur kedua yang dapat disebut dengan *feature map* 2 akan mengalami proses normalisasi fitur.

Proses normalisasi fitur dilakukan karena *feature map* memiliki bentuk data 2D dimana bentuk ini tidak sesuai sebagai *input* untuk *fully connected layer* dan proses pemetaan kelas. Agar *feature map* dapat dipetakan dan diklasifikasikan pada *fully connected* *layer,* maka *feature map* cukup melalui proses *flatten* atau perataan dimensi 2D sehingga data dari yang sebelumnya berbentuk 2D menjadi berbentuk 1D.

Setiap elemen dalam *feature map* 1D tidak akan secara langsung digunakan sebagai *input* dalam *layer* klasifikasi. *Feature map* 1D akan diproses oleh beberapa lapis *hidden layer (gambar 2.4)* berbentuk *fully connected* yang dikaitkan dengan *layer* aktivasi Relu (persamaan 2.9). Proses ini menggunakan *feature map* 1D sebagai *input* lalu memetakannya menggunakan persamaan 2.6 pada *hidden layer* sehingga menghasilkan suatu *output* berbentuk vektordengan panjang yang sesuai dengan jumlah kelas klasifikasi (jenis bunga jepun).

*Output* dari *hidden layer* akan dipetakan pada *layer* klasifikasi berbentuk *layer fully connected* dengan menggunakan persamaan 2.6. *Layer* ini memiliki *output* berbentuk vektordengan panjang yang sesuai dengan jumlah kelas klasifikasi (jenis bunga jepun), oleh karena itu masing-masing kelas label akan memiliki nilai skalaryang disimpan oleh vektor tersebut. Nilai skalar dari masing-masing kelas dalam vektor *output* masih bersifat individu sehingga dibutuhkan *layer* aktivasi dengan fungsi *softmax.* Vektoryang memuat nilai skalar masing masing kelas akan diubah oleh *layer* aktivasi *softmax* menggunakan persamaan 2.14sehingga nilai skalar masing-masing kelas berubah menjadi nilai probabilitas kelas dan jika seluruh elemen vektor dijumlahkan maka akan menghasilkan 1*.* Nilai probabilitas kelas terbesar dapat dikatakan sebagai ouput label kelas dari model CNN.

Untuk proses penguraian ukuran citra dari *input* hingga setelah manghasilkan *output* label kelas akan dijelaskan pada tabel 3.2 dalam satuan px (*pixel*).

Tabel 3.2 Detail Penguraian Citra

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Urutan *Layer* | | *Input* (*depth*) | Ukuran (*input*) | Kernel | Stride | Padding | Ukuran (*output*) |
| *In* | Citra gambar | 3 | 64 x 64 | - | - | - | 64 x 64 |
| 1 | Conv 1 | 6 | 64 x 64 | 5 x 5 | 1 | 2 | 64 x 64 |
| 2 | Max pool 2 | 6 | 64 x 64 | 2 x 2 | 2 | - | 32 x 32 |
| 3 | Conv 3 | 6 | 32 x 32 | 3 x 3 | 1 | 0 | 30 x 30 |
| 4 | Max pool 4 | 6 | 30 x 30 | 2 x 2 | 2 | - | 15 x 15 |
| 5 | FC 5 | 1 | 1350 | 675 | - | - | 675 |
| 6 | FC 6 | 1 | 675 | 84 | - | - | 84 |
| 7 | FC (*output*) | 1 | 84 | 4 | - | - | 4 |

Pada setiap pengenalan nantinya akan didapatkan nilai *value loss / error* dari model CNN yang dihasilkan. *Value loss* ini didapat dari perbandingan probabilitas masing masing kelas *output* dengan label atau probabilitas seharusnya. Perbandingan ini menggunakan persamaan *cross entropy error* yaitu persamaan 2.15.

Contoh data yang terlibat dalam perhitungan *cross entropy error*:

Probabilitas kelas [0.1, 0.1, 0.1, 0.7]

Target label [0, 0, 0, 1]

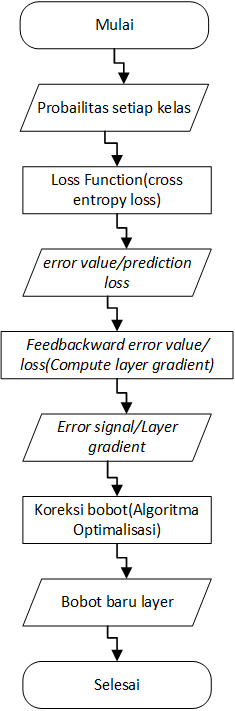
Berdasarkan nilai *value loss / error* tersebut maka proses pelatihan akan berhenti apabila nilai *value loss* sudah mencapai dan hampir melewati nilai *threshold* yang digunakan atau batas *epoch* pelatihan telah tercapai.

Setelah menyelesaikan proses *feedforward*, langkah selanjutnya merupakan *feed backward*. Selama proses *feed backward* ini juga dilakukan proses *update* dan penyesuaian bobot pada setiap *layer* model CNN yang memiliki bobot. Penyesuaian bobot dilakukan secara mundur dari *fully connected layer* hingga *convolutional layer* pertama penyesuaian bobot sendiri berpatokan dengan *value loss / error* yang didapat dari perhitungan di dalam *fully connected layer* oleh persamaan 2.15*.*

Jika dalam tahap *feedforward layer* ditumpuk dari bawah ke atas atau dari *input layer* hingga *layer* klasifikasi, maka dalam *backpropagation layer* ditumpuk dari atas ke bawah atau dari *layer* klasifikasi hingga ke *input layer*. Dimana *error signal* akan bergerak mundur sesuai dengan tumpukan *layer (jika feedforward* urutannya berbentuk *layer* ke *layer* aktivasi maka pada tahap *backpropagation* urutannya berbentuk *layer* aktivasike *layer)*. Dengan menggunakan persamaan 2.24 hingga 2.30 dan melibatkan *error signal,* maka akan dihasilkan arah gradien untuk masing masing *layer* dengan atribut *weight* dan *bias*. Setelah mengetahui arah dari gradienuntuk masing masing *layer* yang memiliki atribut *weight* dan *bias,* maka proses *backpropagation* telah selesai.

Selanjutnya proses koreksi *layer* dilakukan dengan menggunakan RMSprop dan Adam. Untuk setiap gradiendari parameter *weight* dan *bias* yang ada di dalam jaringan akan melalui proses koreksi menggunakan RMSprop (persamaan 2.22 dan persamaan 2.23), ataupun Adam (persamaan 2.16 hingga persamaan 2.21).

Dengan melakukan penyesuaian terhadap bobot-bobot (*weight* dan *bias*) pada *layer*, nantinya didapatkan nilai bobot yang optimal untuk kasus pengenalan citra bunga jepun. Model *Convolutional Neural Network* (CNN) yang dihasilkan dari tahap pelatihan ini selanjutnya akan disimpan dan digunakan pada tahap *testing* atau pengujian.



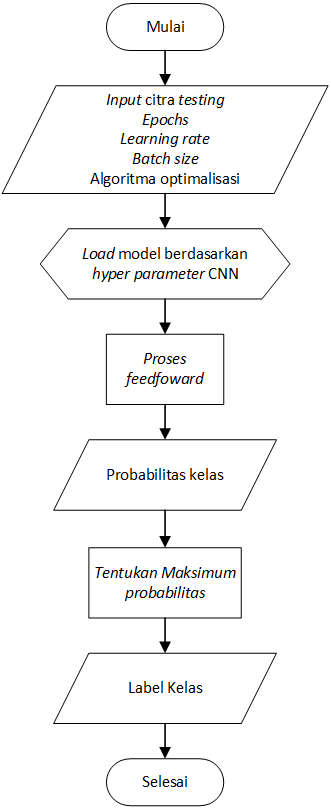
Gambar 3.8 Diagram Alir Tahap *Feed backward*.

Pelatihan ini akan dilakukan pada empat kelompok varietas Jepun Bali jenis *Plumeria sp.* untuk masing masing skenario kombinasi *tuning parameter* dan algoritma optimalisasi. Proses *feedforward* dan *feed backward* akan terus diulang untuk setiap *input* citra. Proses pelatihan akan selesai jika parameter *epoch* sudah memenuhi kondisi berhenti. Alur dari tahap ini dapat dilihat pada gambar 3.8.

### Tahap Pengujian Model CNN

Pada Tahap Pengujian berfungsi untuk menguji apakah model CNN dari hasil tahap pelatihan dapat memberikan *output* berupa label berdasarkan citra *input* bunga jepun*.* Dalam tahap ini akan dapat diketahui jenis bunga jepun apa yang menjadi citra *input* untuk setiap kasus.

Model dari CNN yang telah dibentuk pada tahap pelatihan untuk masing masing skenario parameter pelatihan dan algoritma *optimalisasi* akan digunakan untuk mengenali jenis bunga jepun yang menjadi citra *input*.



Gambar 3.9 Diagram Alir Tahap Pengujian

Tahap ini diawali dengan proses *input* parameter pelatihan yang meliputi *epoch, learning rate,* algoritma optimalisasi *,*dan *batch size*. Selanjutnya objek CNN akan memuat model yang sesuai dengan skenario parameter *input*. Setelah itu object akan mencari model yang sesuai dengan skenario parameter *input*. Setelah objek CNN berhasil memuat model sesuai dengan parameter *input*, citra *input* untuk skenario pengujian akan dimasukan dan diproses secara *feedforward* seperti proses *Feedforward* pada tahap pelatihandalam model CNN*.* Setelah melalui proses *feedforward,* model CNN akan memberikan *output* berupa label yang dikenali dari citra bunga jepun. Diagram alir dari proses pengujian model CNN ditampilkan pada gambar 3.9.

# BAB IV IMPLEMENTASI DAN HASIL

## Tujuan Penelitian

Pada bab ini akan diuraikan mengenai hasil yang didapatkan dalam penelitian serta pembahasan berkaitan tahapan-tahapan yang dilakukan dalam penelitian meliputi, tahap *preprocessing,* pelatihan atau *training,* danpengujian atau *testing* model jaringan syaraf dari metode *Convolutional Neural Network* (CNN).

## Spesifikasi Hardware dan Software

Perangkat yang digunakan dari dan sebagai alat bantu dalam proses implementasi hingga *running* sistem dalam penelitian ini adalah sebagai berikut:

1. Hardware

Dari segi perangkat keras, perangkat yang digunakan merupakan laptop dengan spesifikasi, Processor Hexa-Core 2.60 GHz - 4.50 GHz, Memori RAM 8GB 2666 MHz.

1. Ekosistem *software* dan *tools*

Sistem dibangun menggunakan bahasa pemrograman python versi Python 3.8.3 tags/v3.8.3:6f8c832, May 13 2020, dengan ekosistem *virtual environment* yang terpisah dari ekosistem python utama sistem operasi perangkat. Adapun *tools* dan *library* yang digunakan dijabarkan pada tabel 4.1.

Tabel 4.1 *Tools* dan *library*

|  |  |
| --- | --- |
| *Tools* dan *Library* | Fungsi |
| ipykernel==5.3.4  ipython==7.19.0 | *Python kernel compiler* |
| matplotlib==3.3.3 | memunculkan gambar |
| numpy==1.18.5 | Operasi array yang meliputi operasi matrikss |
| opencv-python==4.4.0.46 | *image loader* |
| scikit-learn==0.23.2  scipy==1.4.1 | operasi konvolusi 2 matrikss. *Library* ini memiliki waktu komputasi lebih cepat daripada operasi yang ditulis menggunakan *native* python |
| spyder==4.2.0  spyder-kernels==1.10.0 | code editor dan kernel |
| einops==0.3.0 | Rotasi struktur matrikss |

## Tahap Implementasi

## Implementasi Tahap *Preprocessing*

Proses pertama yang dilakukan pada tahap *preprocessing* adalah *cropping*. Proses *cropping* dilakukan untuk menghilangkan area pada citra yang tidak memuat objek bunga jepun (mengambil *point of interest* dari citra yang memuat object bunga jepun). Proses *cropping* inidilakukan secara *manual* sebelum citra dapat diproses oleh sistem. Gambar 4.1 menunjukan ilustrasi dari proses *cropping* yang telah dilakukan.



Gambar . Proses *preprocessing* data citra

Pemotongan pada citra dalam dataset dilakukan berdasarkan area citra yang memuat objek bunga jepun, dapat dilihat pada gambar 4.1 citra kiri memuat beberapa informasi pada *background* yang tidak digunakan dalam proses klasifikasi. Proses *cropping* menggunakan rasio 1:1 (1 banding 1) agar citra hasil *cropping* memiliki dimensi sama sisi atau segi empat dengan hanya memuat objek bunga jepun di dalamnya.

Proses kedua dari tahap *preprocessing* yaitu resize. Seluruh citra dalam dataset di*resize* ke dalam ukuran 64 kali 64 px. Tahap *preprocessing* ini dilakukan pada 100 buah citra dari masing-masing kelas sehingga seluruh citra memiliki resolusi seragam.

Proses ketiga merupakan proses konversi ruang warna dari citra yang telah dimuat dalam sistem. Proses konversi ruang warna ini bersifat *optional* yang hanya dilakukan jika citra yang dimuat oleh sistem tidak menggunakan format ruang warna RGB. Dalam penelitian ini, *library* Opencv dari Bahasa pemrograman Python yang digunakan sebagai *tools* untuk memuat citra dataset, secara *default* menggunakan format warna BGR oleh karena itu ruang warna dari citra yang telah dimuat harus diubah dahulu ke dalam format ruang warna RGB.

Proses keempat merupakan proses menyesuaikan matrikss dari citra. Proses ini perlu dilakukan karena spesifikasi program menerima dataset citra sebagai *input* dengan urutan [*image, channel, width, height*], sedangkan format dari matrikss untuk setiap citra yang dimuat adalah [*width, height, channel*].

Gambar 4.2 merupakan penggalan *code* yang menangani proses ketiga dan proses keempat.

|  |
| --- |
| for f in filenames:  #load images  file\_path = os.path.join(dirpath, f)  #print(file\_path)  img = cv2.imread(file\_path)  img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  img = rearrange(img, ' h w c -> c h w ')  arr\_img.append(img)  arr\_label.append(i-1)  self.count+=1 |

Gambar 4.2 Potongan *Code Preprocessing*

Proses tiga dan empat dalam tahap *preprocessing* dilakukan dengan menggunakan fungsi bantuan yang telah disediakan oleh *library* Opencv dan einops dari bahasa pemrograman python yang dituliskan seperti pada gambar 4.2.

Data citra tersebut kemudian dibagi secara acak untuk data latih dan data uji. Sebanyak 75 buah data pada setiap kelasnya digunakan sebagai data latih, selanjutnya sebanyak 20 buah data pada setiap kelasnya digunakan sebagai data uji.

## Implementasi Tahap Pelatihan

Proses pelatihan atau *training* diawali dengan inisiasi model CNN dan *layer* yang terdapat dalam model CNN. Beberapa jenis *layer* yang digunakan yakni, *convolution layer*, *pooling layer*, dan *fully connected layer*. *Activation layer* yakni, *Relu*, dan *softmax*. Fungsi tujuan yaitu *crossentropy* *error*. Seluruh tipe *layer* yang dijabarkan sebelumnya akan dibentuk ke dalam beberapa objek *layer* dalam sistem sehingga secara keseluruhan model CNN akan memuat beberapa *layer* sebagai berikut: conv1, relu1, pool2, conv3, relu3, pool4, fc5, relu5, fc6, relu6, *output*, dan *softmax\_crossentropy*.

Gambar 4.3 merupakan penggalan fungsi *constructor* dari model CNN yang menghandle proses inisialisasi setiap objek *layer*. Proses ini menerima salah satu *Hyperparameter filter* dari *Convo Layer* dan *pooling layer* seperti yang telah disajikan pada tabel3.2,yakni ukuran *window* atau ukuran *kernel* dan *stride* atau besar ukuran pergeseran*.*

|  |
| --- |
| def \_\_init\_\_(self, t\_*input*, t\_*output*, v\_*input*, v\_*output*, method, *epoch*s):  # Conv *Layer*-1  conv1 = CONV\_*LAYER*((6, 64, 64), (6, 3, 5, 5), (2304, 13824), pad=2, stride=1,)  relu1 = RELU\_*LAYER*()  # Sub-sampling-1  pool2 = MAX\_POOL\_*LAYER*(stride=2)  # Conv *Layer*-2  conv3 = CONV\_*LAYER*((6, 32, 32), (6, 6, 3, 3), (576, 3456), pad=0, stride=1, )  relu3 = RELU\_*LAYER*()  # Sub-sampling-2  pool4 = MAX\_POOL\_*LAYER*(stride=2)  # Fully Connected-1  fc5 = FC\_*LAYER*(675, (1350, 675), )#filename=b\_dir+"fc6.npz")  relu5 = RELU\_*LAYER*()  # Fully Connected-2  fc6 = FC\_*LAYER*(84, (120, 84), )#filename=b\_dir+"fc8.npz")  relu6 = RELU\_*LAYER*()  # Fully Connected-3  *output* = FC\_*LAYER*(4, (84, 4), )#filename=b\_dir+"fc10.npz")  softmax\_crossentropy = Activation\_Softmax\_Loss\_CategoricalCrossentropy()  self.*layer*s = [conv1, relu1, pool2, conv3, relu3, pool4, fc5, relu5, fc6, relu6, *output*, softmax\_crossentropy] |

Gambar 4.3 *Code Constructor* model *CNN*

Selanjutnya dataset citra latih akan dikolompokan ke dalam beberapa kelompok *minibatch.* Jumlah *minibatch* dihitung dengan cara jumlah seluruh data citra *training* dibagi dengan ukuran *minibatch.* Setelah itu tahap *training* dijalankan dalam bentuk dua rangkap perulangan. Perulangan pertama dilakukan dengan menggunakan *epoch* sebagai batas penghentian. Jika siklus pelatihan seluruh data latih telah dilakukan sebanyak maksimum jumlah *epoch,* makaproses pelatihan akan dihentikan. Perulangan kedua berada di dalam perulangan pertama. Perulangan ini akan mengulang tahap pelatihan (mencakup *feedforward, feed backward atau backpropagation,* dan *parameter update*) untuk setiap kelompok *minibatch.* Setelah seluruh *minibatch* telah melakukan koreksi bobot maka model akan divalidasi menggunakan dataset *testing* dimana ini bertujuan untuk melihat kemampuan dari model untuk mengenali suatu data objek citra beserta kelasnya di luar dari dataset *training.* Jika seluruh kelompok *minibatch* telah menyelesaikan prose pelatihan, maka satu siklus pelatihan telah berakhir dan *epoch* pelatihan akan berganti. Gambar 4.4 merupakan penggalan kode dalam fungsi *LENET5\_TRAIN* yang menghandle proses pelatihan model.

|  |
| --- |
| num\_batches = int(np.ceil(X\_train.shape[0] / batch))  X\_batches = zip(np.array\_split(X\_train, num\_batches, axis=0), np.array\_split(Y\_train, num\_batches, axis=0))  for ep in range(*epoch*s):  self.optimizer.pre\_update\_params()  print("*Epoch*: ", ep)  for x, y in X\_batches:  predictions, loss = LENET5.feedForward(x, self.*layer*s, y)  if len(y.shape) == 2 and len(predictions.shape) == 2 :  temp\_y = np.argmax(y, axis=1)  temp\_predictions = np.argmax(predictions, axis=1)  acc = np.sum(np.equal(temp\_y,temp\_predictions))/len(temp\_y)  else :  acc = np.mean(predictions==y)  LENET5.backpropagation(y, self.*layer*s)  LENET5.update\_parameters(self.*layer*s, self.optimizer ,  alpha, method)  print("Step::", "\tAcc::", "\tLoss:: ", "\tWeight\_sum:: ")  self.optimizer.post\_update\_params()  XY = list(zip(X\_train, Y\_train))  np.random.shuffle(XY)  new\_X, new\_Y = zip(\*XY)  X\_batches = zip(np.array\_split(new\_X, num\_batches, axis=0),  np.array\_split(new\_Y, num\_batches, axis=0)) |

Gambar 4.4 *Code* Tahap pelatihan (*training*)

Pada *method Feedforward* kelas utama model CNN, terdapat beberapa pengecekan kondisi dari tipe kelas objek *layer* dikarenakan setiap jenis *layer* memberikan bentuk *output* yang berbeda-beda. Seperti yang sudah dijelaskan pada bab 3, terdapat proses pengecekan *input* dan normalisasi *input* dikarenakan *output* dari Convolutional *layer* dan *max pooling layer* berbentuk 4D meliputi [*sampel, depth, height, width*], sedangkan *input* dari *fully connected* *layer* berbentuk 2D yang meliputi [*sampel, feature*]. Gambar 4.5 merupakan potongan fungsi kode *feed foward* dari model CNN.

Tabel 4.2 menampilkan contoh salah satu dari enam *feature map* citra bunga jepun hasil ekstraksi fitur pada *pooling layer* ke dua, dengan resolusi *feature map* sebesar 15 kali 15 satuan yang belum mengalami normalisasi. Berdasarkan tabel 3.2, satu citra akan menghasilkan data fitur sebanyak 1350 dalam satu baris vektor yang selanjutnya akan diproses oleh *fully conntected layer.*

Tabel 4.2 *Feature map* dan tampilan dalam citra

|  |  |
| --- | --- |
| *Feature map* | *Citra* |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 17 | 21 | 23 | 25 | 23 | 15 | 5 | 0 | 19 | 41 | 53 | 65 | 65 | 40 | 30 | | 48 | 60 | 60 | 56 | 50 | 39 | 16 | 10 | 30 | 41 | 52 | 69 | 70 | 46 | 30 | | 52 | 61 | 60 | 57 | 50 | 43 | 30 | 18 | 26 | 31 | 51 | 70 | 72 | 49 | 30 | | 42 | 47 | 46 | 44 | 39 | 29 | 26 | 19 | 15 | 23 | 47 | 64 | 66 | 47 | 30 | | 20 | 27 | 27 | 22 | 18 | 9 | 6 | 7 | 3 | 10 | 27 | 39 | 42 | 39 | 36 | | 2 | 13 | 21 | 16 | 3 | 0 | 0 | 0 | 0 | 5 | 19 | 30 | 39 | 49 | 50 | | 15 | 30 | 35 | 28 | 10 | 0 | 0 | 0 | 0 | 1 | 7 | 23 | 44 | 58 | 61 | | 44 | 52 | 47 | 26 | 8 | 0 | 0 | 0 | 0 | 1 | 16 | 41 | 60 | 69 | 70 | | 59 | 60 | 48 | 23 | 3 | 0 | 0 | 0 | 0 | 10 | 32 | 56 | 69 | 71 | 69 | | 65 | 62 | 48 | 23 | 7 | 8 | 3 | 0 | 2 | 19 | 46 | 61 | 69 | 68 | 54 | | 67 | 63 | 46 | 23 | 30 | 30 | 18 | 10 | 8 | 20 | 44 | 53 | 53 | 46 | 34 | | 67 | 62 | 45 | 40 | 37 | 41 | 41 | 36 | 32 | 27 | 34 | 35 | 30 | 24 | 19 | | 56 | 53 | 45 | 42 | 32 | 49 | 56 | 54 | 53 | 50 | 43 | 39 | 38 | 40 | 41 | | 37 | 37 | 37 | 34 | 28 | 45 | 58 | 60 | 64 | 65 | 63 | 48 | 51 | 54 | 54 | | 33 | 32 | 30 | 30 | 29 | 30 | 47 | 57 | 64 | 69 | 67 | 55 | 54 | 55 | 55 | |  |

|  |
| --- |
| def feedForward(X, *layer*s, y\_*true*):  inp = X  wsum = 0  y\_*true* = y\_*true*  for *layer* in *layer*s:  if isinstance(*layer*, FC\_*LAYER*) and len(inp.shape) == 4:  inp, ws = *layer*.forward(inp.reshape(inp.shape[0], inp.shape[1]\*inp.shape[2]\*inp.shape[3]))  elif isinstance(*layer*, Activation\_Softmax\_Loss\_CategoricalCrossentropy):  loss, ws = *layer*.forward(inp,y\_*true*)  inp = *layer*.*output*  else:  inp, ws = *layer*.forward(inp)  wsum += ws  return inp, loss, wsum |

Gambar 4.5 *Code Feedforward* dari model CNN

Setiap kelas *layer* memiliki *method* atau fungsi *Feedforward* yang nantinya akan dipanggil oleh kelas utama model CNN. Gambar 4.6 merupakan potongan kode dari fungsi *feedforward* untuk *layer* dengan jenis objek *convolutional layer* yang mengimplementasikan persamaan 2.2 hingga 2.5. *Output* dari potongan kode program ini merupakan *feature map* hasil operasi konvolusi antara *input matrikss* dan *kernel konvolusi* (gambar 2.6 dan gambar 2.10).

|  |
| --- |
| def forward(self, X):  self.feature\_map = np.zeros([N, K, conv\_h, conv\_w])  X\_padded = np.pad(X, ((0,0), (0,0), (pad\_len, pad\_len), (pad\_len, pad\_len)), 'constant')  if stride == 1:  kernel\_180 = np.rot90(self.kernel, 2, (2,3))  for img in range(N):  for conv\_depth in range(K):  for inp\_depth in range(D):  self.feature\_map[img, conv\_depth] += scipy.signal.convolve2d(X\_padded[img, inp\_depth], kernel\_180[conv\_depth, inp\_depth], mode='valid')  self.feature\_map[img, conv\_depth] += self.bias[conv\_depth]  else:  for img in range(N):  for conv\_depth in range(K):  for h in range(0, H + 2\*pad\_len - K\_H + 1, stride):  for w in range(0, W + 2\*pad\_len - K\_W + 1, stride):  self.feature\_map[img, conv\_depth, h//stride, w//stride] = \  np.sum(np.multiply(X\_padded[img, :, h:h+K\_H, w:w+K\_W], self.kernel[conv\_depth,:,:,:])) + self.bias[conv\_depth]  return self.feature\_map, np.sum(np.square(self.kernel)) |

Gambar 4.6 *Code* *Feedforward* dari *Convolutional Layer*

*Feature map* yang ditampilkan pada tabel 4.3 hingga tabel 4.6 merupakan *feature map* convo *layer* I dan II yang sudah melalui fungsi aktivasi Relu, dimana fungsi ini akan menghilangkan nilai negatif dari *feature map* sekaligus menentukan data mana yang dianggap penting untuk diproses pada tahap selanjutnya. Terlihat bahwa beberapa *feature map* gelap atau mati. Ini artinya *kernel* yang menghasilkan *feature map* tersebut gagal mendeteksi atau mengambil data dari *input*. Keadaan ini disebut dengan istilah *dead neuron / dead kernel.* Disisi lain, kernel yang berhasil mendeteksi objek bunga jepun untuk seluruh kelas memberikan suatu *output* yang berupa 1 lapis matrikss 2 dimensi yang disebut *feature*.

Tabel 4.3 *Feature map* Convo *Layer* I dan II

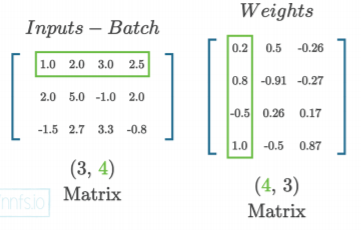
|  |  |  |
| --- | --- | --- |
| Citra *Input* | *Feature map* Conv 1 | *Feature map* Conv 2 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Jika ditelisik lebih jauh, terdapat kesamaan informasi yang disajikan oleh masing masing *feature map* untuk setiap kelas bunga jepun. Informasi ini berupa eliminasi atau hilangnya warna kuning dari setiap *feature* yang dihasilkan. Proses eliminasi ini dilakukan dengan cara memberikan nilai nol pada daerah yang mengandung warna kuning dari *input,* hal ini menyebabkan daerah tersebut menjadi gelap pada seluruh *feature map*. Ini mengidentifikasikan bahwa informasi yang dimiliki oleh seluruh kelas bunga jepun (informasi warna kuning) dapat dianggap sebagai suatu fitur warna yang tidak penting, sehingga dihilangkan oleh *kernel* dari *convolutional layer.*

Gambar 4.7 merupakan potongan kode dari fungsi atau *method* *feedforward* untuk *layer* dengan jenis objek kelas *fully connected layer* yang mengimplementasikan persamaan 2.1 dan 2.6.

|  |
| --- |
| def forward(self, X):  kernel, bias = self.kernel, self.bias  self.cache = (X, kernel, bias)  self.activations = np.dot(X, kernel) + bias  return self.activations, np.sum(np.square(self.kernel)) |

Gambar 4.7 *Code* *Feedforward* dari *Fully Connected Layer*



Gambar 4.8 Ilustrasi *Input* dan *Weight Dalam Operasi Dot Matrikss*

Menurut Kinsley (2019) jika persamaan 2.1 dan 2.6 diterapkan pada setiap elemen *input* dan parameter *weight,* maka akan melibatkan banyakvariabel seiring dengan banyaknya interkoneksi dari neuron untuk suatu model yang lebih kompleks. Oleh karenanya operasi *dot()* matrikss jauh lebih efisien diterapkan karena implementasi dalam kode yang jauh lebih ringkas. *Input* dari program ini berupa matrikss yang memuat *list* setiap vektor dari sampel. Gambar 4.8 merupakan ilustrasi dari operasi *dot().*

Gambar 4.9 merupakan potongan kode dari fungsi atau *method* *feedforward* untuk *layer* dengan jenis objek kelas *pooling layer.*

|  |
| --- |
| def forward(self, X):  factor = self.factor  N, D, H, W = X.shape  #assert H%factor == 0 and W%factor == 0  self.cache = [X, factor]  self.feature\_map = X.reshape(N, D, H//factor, factor, W//factor, factor).max(axis=(3,5))  #assert self.feature\_map.shape == (N, D, H//factor, W//factor)  return self.feature\_map, 0 |

Gambar 4.9 *Code* *Feedforward* dari *Fully Connected Layer*

Programinimengembalikan *output* berupa *feature map* yang memuat fitur dari citra *input*. *Feature map* ini merupakan *feature map* yang sama dengan *feature map* *output* dari *convolutional layer* di belakangnya dengan resolusi tereduksi. Resolusi yang tereduksi dikarenakan *pooling layer* hanya mengambil elemen maksimum untuk setiap daerah spatial dalam *window* atau *kernel* (gambar 2.12) untuk masing-masing *feature map* hasil ekstraksi dari *convolution kernel* dancitra.

Tabel 4.4 hingga 4.7 menyajikan perbandingan antara *feature map* yang merupakan *output* dari masing masing Convo *layer* I dan II yang sudah melalui fungsi aktivasi Relu, dengan *feature map* tereduksi yang merupakan *output* dari *Max Pool layer.* Dapat dilihat pada tabel 4.4 hingga 4.7, walaupun *feature map* memiliki resolusi yang tereduksi, *feature map* masih dapat menyajikan informasi pixel dari citra bunga jepun.

Tabel 4.4 *Feature map Max Pool layer* I dan II untuk Jepun Bali

|  |  |
| --- | --- |
| Convo 1 Fmap | Pool 1 Fmap |
|  |  |
| Convo 2 Fmap | Pool 2 Fmap |
|  |  |

Tabel 4.5 *Feature map Max Pool layer* I dan II untuk Jepun Cendana

|  |  |
| --- | --- |
| Convo 1 Fmap | Pool 1 Fmap |
|  |  |
| Convo 2 Fmap | Pool 2 Fmap |
|  |  |

Tabel 4.6 *Feature map Max Pool layer* I dan II untuk Jepun Merah Muda

|  |  |
| --- | --- |
| Convo 1 Fmap | Pool 1 Fmap |
|  |  |
| Convo 2 Fmap | Pool 2 Fmap |
|  |  |

Tabel 4.7 *Feature map Max Pool layer* I dan II untuk Jepun Sudamala

|  |  |
| --- | --- |
| Convo 1 Fmap | Pool 1 Fmap |
|  |  |
| Convo 2 Fmap | Pool 2 Fmap |
|  |  |

Gambar 4.10 merupakan potongan kode dari fungsi atau *method* *feedforward* untuk *layer* aktivasi dengan jenis objek kelas *Relu* yang mengimplementasikan persamaan 2.9. program ini akan mengembalikan *output* berupa *input* tanpa elemen yang bernilai negatif. Program ini menggunakan fungsi bantuan dari *library numpy* dimana fungsi yang dimaksud adalah fungsi *maximum.* Fungsi maksimum membandingkan dan mengembalikan nilai maksimum berdasarkan *parameter input* dan *parameter* batas yang diberikan.

|  |
| --- |
| def forward(self, X):  self.cache = X  self.*output* = np.maximum(X, 0)  return self.*output*, 0 |

Gambar 4.10 *Code* *Feedforward* dari *ReLu Layer*

Gambar 4.11 merupakan potongan kode dari fungsi atau *method* *feedforward* untuk *layer* aktivasi dengan jenis objek kelas *softmax* yang mengimplementasikan persamaan 2.14 dimana program ini memberikan *output* berupa probabilitas untuk masing-masing kelas dari sampel.

|  |
| --- |
| def forward(self, *input*s):  exp\_values = np.exp(*input*s - np.max(*input*s, axis=1, keepdims=*True*))  probabilities = exp\_values / np.sum(exp\_values, axis=1, keepdims=*True*)  self.*output* = probabilities  return self.*output*, 0 |

Gambar 4.11 *Code* *Feedforward* dari *Softmax Layer*

Program ini menggunakan fungsi bantuan dari *library numpy* dimana fungsi yang dimaksud adalah fungsi *exp* yang mengembalikan nilai hasil perhitungan *exponent* *e* dari matrikss *input.* Menurut Kinsley (2019), fungsi *exponensial* yang digunakan dalam fungsi aktivasi *softmax* dapat menyebakan “*exploding-value*” atau nilai yang terlalu besar sehingga membanjiri memori dan menyebabkan *RuntimeWarning: overflow encountered in exp inf.* Diketahui suatu fungsi *eksponential* cenderung mendekati 0 jika x mengarah ke negatif *infinity* dan 1 jika x bernilai 0. Untuk itu *matrikss sampel* akan melalui proses normalisasi dengan cara mengurangi setiap elemennya dengan elemen terbesar dari baris tersebut.

Contoh:

*Input* = [1, 2, 3]

Normalisasi = [-2, -1, 0]

Exp = [0.13533528, 0.36787944, 1]

Probabilitas hasil normalisasi = [0.09003057 0.24472847 0.66524096]

Gambar 4.12 potongan kode dari fungsi atau *method* *feedforward* untuk *layer* objektif dengan jenis objek kelas *cross entropy error* yang mengimplementasikan persamaan 2.15 dimana program ini akan memberikan *output* berupa rata-rata *loss* dari *batch* *input*.

|  |
| --- |
| class Loss:  def calculate(self, *output*, y):  sampel\_losses = self.forward(*output*, y)  data\_loss = np.mean(sampel\_losses)  return data\_loss  # Cross-entropy loss  class Loss\_CategoricalCrossentropy(Loss):  def forward(self, y\_pred, y\_*true*):  sampels = len(y\_pred)  y\_pred\_clipped = np.clip(y\_pred, 1e-7, 1 - 1e-7)  if len(y\_*true*.shape) == 1:  correct\_confidences = y\_pred\_clipped[range(sampels),y\_*true*]  elif len(y\_*true*.shape) == 2:  multiply = np.multiply(y\_pred\_clipped, y\_*true*)  correct\_confidences = np.sum(multiply, axis=1)  negatif\_log\_likelihoods = 1\*(-np.log(correct\_confidences))  return negatif\_log\_likelihoods |

Gambar 4.12 *Code* *Feedforward* dari *Categorial Cross Entropy*

Program ini menggunakan fungsi bantuan dari *library numpy* dimana fungsi yang dimaksud adalah fungsi *log. Output dari Softmax* merupakan *input* dari fungsi ini, dimana nilai untuk setiap elemennya berada di antara 0 sampai 1. Ada kemungkinan bahwa model CNN menghasilkan suatu label yang bernilai *confident* 1 dan mengakibatkan label lainnya bernilai 0. Bisa saja nilai 1 ini bukan merujuk kepada nilai yang bukan merupakan *target* atau *true class.* Hal ini menyebabkan operasi logaritma bernilai 0 dan memberikan *warning RuntimeWarning: divide by zero encountered in log inf.* Menurut Kinsley (2019), permasalahan ini dapat diselesaikan dengan menambah batas bawah dari rentang nilai dengan nilai yang sangat kecil (*1e-7*) sehingga tidak memiliki efek yang signifikan, sedangkan merubah batas atasnya menjadi (1-*1e-7*). Jika menggunakan (1+*1e-7)* akan menimbulkan permasalahan baru dimana jika model memberikan prediksi yang penuh atau bernilai 1, maka nilai *error* atau *loss* akan bernilai negatif dan bukannya 0. Jika menggunakan batas atas menjadi 1, maka model akan mengembalikan nilai error 0 negatif yang dapat memberikan masalah baru dalam proses komputasi.

Potongan kode pada gambar 3.12 perhitungan loss atau error ini bisa menerima dua bentuk *true* *value* atau *true target.*  Jenis pertama adalah *true target* dalam bentuk *array* 1 Dimensi yang dimana setiap elemen dari *array* merujuk pada kelas yang benar. Sebagai contoh:

Softmax *input*:

[[0.7, 0.1, 0.2], sampel ke 0

[0.1, 0.5, 0.4], sampel ke 1

[0.02, 0.9, 0.08]] sampel ke 2

class\_targets or y\_*true* = np.array([0, 1, 1])

print(distribution[targ\_idx])

[0.7, 0.1, 0.2] -> [0]

Maka untuk sampel distribusi 0, kelas yang benar menurut *array class\_target* adalah kelas ke 0 yang bernilai 0.7 dan hanya nilai ini yang akan diproses pada perhitungan selanjutnya. Nilai ini menandakan tingkat kepercayaan prediksi dari model CNN.

Satu lagi adalah *true target* yang berbentuk *one hot* vektoratau *class target* berbentuk 2 dimensi dimana untuk setiap baris dalam matrikss hanya akan berisi satu nilai 1 yang menandakan posisi dari *true target.* Untuk mensiasati bentuk ini dan mengambil nilai kepercayaan dari model, cukup dengan melakukan operasi perkalian yang secara otomatis akan menghilangkan indeks bernilai 0,contoh:

|  |  |  |
| --- | --- | --- |
| # [[0.7, 0.1, 0.2], x [[1,0,0],  # [0.1, 0.5, 0.4], x [0,1,0],  # [0.02, 0.9, 0.08]] x [0,0,1]] | = [[0.7 0. 0.]  = [0. 0.5 0.]  = [0. 0. 0.08]] | Maka  [0.7 0.5 0.08] |

Pada *method backpropagation* kelas utama model CNN, list yang menyimpan urutan setiap *layer* dalam model CNN akan diproses melalui suatu perulangan yang dimulai dari belakang. Selanjutnya fungsi atau *method* *backward* dari setiap objek *layer* akan dipanggil dan setiap fungsi tersebut akan mengembalikan *output* yang disebut *error signal* atau *loss* pada *layer* ataugradien *layer* terhadap *input*. Berikut ini adalah potongan kode fungsi *backpropagation* dari model CNN.

|  |
| --- |
| def backpropagation(Y, *layer*s):  delta = Y  for *layer* in *layer*s[::-1]:  if isinstance(*layer*, Activation\_Softmax\_Loss\_CategoricalCrossentropy):  delta = *layer*.backward(*layer*.*output*,delta)  else :  delta = *layer*.backward(delta) |

Gambar 4.13 *Code* *Backpropagation* dari Model CNN

Gambar 4.14 merupakan potongan kode *backward method* atau fungsi dari object *layer* bertipe kelas *Loss\_CategoricalCrossentropy*. Kode ini merupakan implentasi dari persamaan 2.35. sesuai dengan istilah dari *backpropagation*, *layer* pertama yang melakukan komputasi *backward* merupakan *layer* fungsi objektif loss yang dalam hal ini merupakan *Cross-entropy loss*.

|  |
| --- |
| # Cross-entropy loss  class Loss\_CategoricalCrossentropy(Loss):  def backward(self, dvalues, y\_*true*):  sampels = len(dvalues)  labels = len(dvalues[0])  if len(y\_*true*.shape) == 1:  y\_*true* = np.eye(labels)[y\_*true*]  self.d*input*s = -y\_*true* / dvalues  self.d*input*s = self.d*input*s / sampels  return self.d*input*s |

Gambar 4.14 *Code* *Backpropagation* dari *Cross Entropy Loss*

Implementasinya cukup sederhana dimana *input* dari fungsi ini merupakan matrikss prediksi dan *true label* dari sampel. Sebelum melakukan perhitungan seperti yang dituliskan pada persamaan 2.35, *true label* dari sampel harus diubah terlebih dahulu ke dalam bentuk *one hot* vektor*. Output* dari program ini merupakan *error signal* yang akan dikirimkan ke *softmax layer.*

*Error signal* dari *Cross-entropy loss* merupakan *input* dari fungsi *feed backward* dari *layer* aktivasi *softmax.* Menurut Kinsley (2019), *Backward method* dari *layer* aktivasi softmax menggunakan konsep jacobian matrikss. Dalam kasus ini, matrikss Jacobian adalah array turunan parsial di semua kombinasi dari kedua vektor *input*. Ingat, persamaan 2.38 menghasilkan dua kasus komputasi berbeda untuk j = k menggunakan persamaan 2.39 dan j ≠ k menggunakan persamaan 2.4. Maka dari itu menghitung turunan parsial dari setiap *output* dari fungsi Softmax sehubungan dengan setiap *input* dilakukan secara terpisah. *Output* dari proses ini merupakan suatu matrikss berbentuk 3D. Bentuk *output* turunan parsial ini tidak dapat diterima oleh *fully connected layer* yang berada di belakangnya.

Turunan dari fungsi Softmax sehubungan dengan salah satu *input*nya mengembalikan vektor turunan parsial (baris dari matrikss Jacobian), karena *input* ini mempengaruhi semua *output*, demikian juga mempengaruhi setiap turunan parsial. Perlunya menjumlahkan nilai dari vektor ini sehingga setiap *input* untuk setiap sampel akan mengembalikan nilai turunan parsial tunggal sebagai gantinya. Karena setiap *input* mempengaruhi semua *output*, vektor yang dikembalikan dari turunan parsial harus dijumlahkan untuk turunan parsial akhir sehubungan dengan *input* ini. operasi ini dapat dilakukan pada setiap matrikss Jacobian secara langsung, menerapkan aturan rantai pada saat yang sama (menerapkan gradien dari fungsi objective atau *loss function*) menggunakan np.dot() - Untuk setiap sampel, ini akan mengambil baris dari Jacobian matrikss dan mengalikannya dengan nilai yang sesuai dari gradien fungsi objektif atau *loss*. Hasilnya, perkalian titik dari masing-masing vektor dan nilai ini akan mengembalikan nilai tunggal, membentuk vektor turunan parsial *sampel-wise* berbentuk matrikss 2D (kumpulan vektor yang dihasilkan) secara *batch* (Kinsley, 2019). Gambar 4.15 merupakan kode dari implementasi metode *backward* pada *layer* *softmax.*

|  |
| --- |
| def backward(self, dvalues):  self.d*input*s = np.empty\_like(dvalues)  for index, (single\_*output*, single\_dvalues) in \  enumerate(zip(self.*output*, dvalues)):  # Flatten *output* array  single\_*output* = single\_*output*.reshape(-1, 1)  jacobian\_matrikss = np.diagflat(single\_*output*) - \  np.dot(single\_*output*, single\_*output*.T)  self.d*input*s[index] = np.dot(jacobian\_matrikss,single\_dvalues)  return self.d*input*s |

Gambar 4.15 *Code* *Backpropagation* dari *Softmax Layer*

Gambar 4.16 merupakan potongan kode program berikut adalah *method* atau fungsi dari *feed backward* untuk objek *layer* aktivasi berjenis kelas *Relu*. Kode ini merupakan implentasi dari turunan fungsi ReLu yang dituliskan oleh persamaan 2.31 dan persamaan 2.32.

|  |
| --- |
| def backward(self, delta):  self.delta\_X = delta.copy()  self.delta\_X[self.cache <=0] = 0  return self.delta\_X |

Gambar 4.16 *Code* *Backpropagation* dari *ReLu Layer*

Implementasinya cukup sederhana dimana *input* dari fungsi ini merupakan matrikss yang memuat turunan parsial atau *error signal* dari *layer* di depannya. Berdasarkan persamaan 2.32 untuk setiap elemen nilai dari turunan parsial *layer* di depannya yang berupa matrikss *input* atau *delta,* menurut persamaan 2.24turunan parsial dari *layer* di depan Relu akan dikalikan dengan turunan dari fungsi relu.Jika nilai *error* *signal* tersebut bernilai lebih kecil atau sama dengan nol, maka nilainya dikalikan 0 yang menghasilkan 0. Untuk nilai yang lebih besar dari nol maka nilai tersebut akan dikalikan satu. *Output* yang berupa turunan parsial dari *layer* Relu akan dikirimkan pada *layer* di belakangnya

Gambar 4.17 merupakan potongan kode program berikut adalah *method* atau fungsi dari *feed backward* untuk objek *layer* aktivasi berjenis kelas *max-pooling*. Turunan dari *max-pooling layer* sama dengan turunan dari fungsi Relu. Kedua *layer* menerapkan konsep fungsi maksimum sehingga secara matematis turunan dari kedua fungsi adalah sama. Kode ini merupakan implentasi dari turunan fungsi *maxpooling* (persamaan 2.31 dan persamaan 2.32).

|  |
| --- |
| def backward(self, delta):  X, factor = self.cache  if len(delta.shape) != 4: # then it must be 2  delta = delta.reshape(self.feature\_map.shape)  fmap = np.repeat(np.repeat(self.feature\_map, factor, axis=2), factor, axis=3)  dmap = np.repeat(np.repeat(delta, factor, axis=2), factor, axis=3)  self.delta\_X = np.zeros(X.shape)  self.delta\_X = (fmap == X) \* dmap  return self.delta\_X |

Gambar 4.17 *Code* *Backpropagation* dari *Max Pool Layer*

Konsepnya adalah dengan mengembalikan resolusi dari *feature map* yang tereduksi. Setiap sampel dalam *feature map* merupakan matrikss 2 dimensi sehingga proses ini dilakukan dengan cara menduplikasi setiap elemen dalam matrikss secara vertikaldan horizontal sebanyak besar pergeseran atau *stride* yang digunakan dalam tahap *feedforward,* proses yang sama juga diterapkan pada turunan parsial yang merupakan *error signal* dari *layer* di depannya. Setelah mengembalikan ukuran setiap sampel dari *feature map* dan turunan parsial, kedua variabel ini dikalikan menurut persamaan 2.27 dan persamaan 2.24. Namun, tidak semua elemen dapat dikalikan menurut persamaan 2.31 dan persamaan 2.32. Elemen dari matrikss *feature map* harus dibandingkan dengan *input* pada tahap *feedforward* sehingga ditemukan posisi elemen *maksimum* yang seharusnya diperhitungkan. Setiap elemen dalam matrikss *error signal* akan dikalikan dengan 1 jika posisinya sama dengan posisi nilai maksimum yang diperhitungkan pada *input X,* jika tidak maka elemen tersebut akan dikalikan dengan 0.

Contoh:

|  |  |  |  |
| --- | --- | --- | --- |
| *X(input)* =  [[2,1,1,1],  [1,2,1,1],  [2,2,3,2],  [3,3,3,4]] | *Fmap* =  [[2,2,2,2],  [2,2,2,2],  [3,3,3,3],  [4,4,4,4]] | dmap =  [[1,1,1,1],  [1,1,1,1],  [1,1,1,1],  [1,1,1,1]] | Hasil =  [[1 0 0 0]  [0 1 0 0]  [0 0 1 0]  [0 0 0 1]] |

Gambar 4.18 merupakan potongan kode program berikut adalah *method* atau fungsi dari *feed backward* untuk objek *layer* berjenis kelas *fully connected layer.* Potongan kode ini merupakan implementasi dari persamaan 2.7, 2.8, 2.27, 2.28, dan 2.30.

|  |
| --- |
| def backward(self, delta):  X = self.cache  self.delta\_X = np.dot(delta, self.kernel.T)  self.delta\_K = np.dot(X.T, delta)  self.delta\_b = np.sum(delta, axis=0)  return self.delta\_X |

Gambar 4.18 *Code* *Backpropagation* dari *FC Layer*

Berdasarkan aturan turunan parsial dalam penjumlahan (persamaan 2.30). Turunan untuk fungsi penjumlahan yang dituliskan pada persamaan 2.28, dapat diabaikan karena untuk setiap nilai turunan parsial yang diberikan adalah 1 sehingga perkalian dengan angka 1 dapat diabaikan. Fokus perhitungan dalam *method feed backward* adalah *learnable parameters* ataubobot(*weight*), *offset* dari fungsi aktivasi yang disebut bias dan gradien fungsi.

Drelu\_xo = dvalue \* (1, if z > 0 else 0.) \* w[0]

Drelu\_wo = x[0] \* (1, if z > 0 else 0.) \* dvalue

Potongan kode diatas adalah patokan dalam perhitungan gradien *parameter layer*. Sisi kiri dari potongan kode diatas merupakan gradien*,* turunan parsial atau *error signal* yang akan dikirimkan pada *layer* di belakangnya. Gradien ini diperoleh dari perhitungan antara turunan fungsi *neuron* (gradien *output* dari fungsi aktivasi) terhadap .

Proses ini menghitung seberapa besar pengaruh yang diberikan oleh suatu *input* dengan melibatkan bobot(*weight*) terhadap *output* keseluruhan dari *layer* dikalikan dengan *error signal* atau gradien dari *layer* di depannya*.* Untuk menghitung seberapa besar gradienatau pengaruh yang diberikan oleh bobot(*weight*) kepada *output layer* adalah dengan menghitung turunan dari fungsi *neuron* terhadap *weight* dengan melibatkan *input* dikalikan dengan *error signal* atau gradien dari *layer* di depannya*.* Untuk menghitung seberapa besar gradienatau pengaruh yang diberikan oleh bias kepada *output layer* adalah dengan menghitung turunan dari fungsi *neuron* terhadap *bias* dimana turunan operasi penjumlahan selalu bernilai 1*.*

Perlu diperhatikan bahwa gradienyang diterima dari *layer* di depannya tersusun dalam bentuk baris per *neuron* sehingga untuk melakukan operasi dot, matriks *input* x dan *weight* harus di transpose untuk menyamakan *inner shape* dari operasi dot sesuai dengan yang dituliskan pada persamaan 2.7 dan persamaan 2.8*.* Pada akhirnya operasi ini akan menghasilkan 3 *output* yang meliputi delta kernel (gradien bobot atau *weight*), delta\_bias (gradien bias), dan delta\_X (gradienfungsi *layer* yang akan dikirimkan pada *layer* di belakangnya).

Gambar 4.19 adalah *method* atau fungsi dari *feed backward* untuk objek *layer* berjenis kelas Convolutional *layer.* Potongan kode ini merupakan implementasi dari persamaan 2.27, 2.28, dan 2.30.

|  |
| --- |
| def backward(self, delta):  X\_padded = np.pad(X, ((0,0), (0,0), (pad\_len, pad\_len), (pad\_len, pad\_len)), 'constant')  delta\_X\_padded = np.zeros(X\_padded.shape)  self.delta\_K = np.zeros(self.kernel.shape)  self.delta\_b = np.zeros(self.bias.shape)  # Delta X  for img in range(N):  for conv\_depth in range(K):  for h in range(0, H + 2\*pad\_len - K\_H + 1, stride):  for w in range(0, W + 2\*pad\_len - K\_W + 1, stride):  delta\_X\_padded[img, :, h:h+K\_H, w:w+K\_W] += np.multiply(delta[img, conv\_depth, h//stride, w//stride], self.kernel[conv\_depth])  if pad\_len > 0:  self.delta\_X = delta\_X\_padded[:, :, pad\_len:-pad\_len, pad\_len:-pad\_len]  else:  self.delta\_X = delta\_X\_padded[:]  # Delta kernel  for img in range(N):  for kernel\_num in range(K):  for h in range(conv\_h):  for w in range(conv\_w):  self.delta\_K[kernel\_num,:,:,:] += np.multiply(delta[img, kernel\_num, h, w], X\_padded[img, :, h\*stride:h\*stride+K\_H, w\*stride:w\*stride+K\_W])  # Delta Bias  self.delta\_b = np.sum(delta, (0,2,3))  return self.delta\_X |

Gambar 4.19 *Code* *Backpropagation* dari *Convolutional Layer*

Seperti pada *layer* berjenis *fully connected,* berdasarkan aturan turunan parsial dalam penjumlahan (persamaan 2.30). Turunan untuk fungsi penjumlahan yang dituliskan pada persamaan 2.28, dapat diabaikan karena untuk setiap kernel konvolusi nilai turunan parsial yang diberikan adalah 1 sehingga perkalian dengan angka 1 dapat diabaikan. Fokus perhitungan dalam *method feed backward* adalah *learnable parameters* atau *kernel* bobot(*weight*), *offset* dari fungsi aktivasi yang disebut bias dan gradien fungsi *layer* konvolusi.

Drelu\_xo = dvalue \* (1, if z > 0 else 0.) \* w[0]

Drelu\_wo = x[0] \* (1, if z > 0 else 0.) \* dvalue

Potongan kode diatas adalah patokan dalam perhitungan gradien *parameter*. Sisi kiri dari potongan kode diatas merupakan gradien*,* turunan parsial atau *error signal* yang akan dikirimkan pada *layer* di belakangnya (Drelu\_xo). Gradien ini diperoleh dari perhitungan antara turunan fungsi *neuron* (gradien *output* dari fungsi aktivasi) terhadap .

Proses ini menghitung seberapa besar pengaruh yang diberikan oleh suatu *input* dengan melibatkan bobot(*weight*) terhadap *output* keseluruhan dari *layer* dikalikan dengan *error signal* atau gradien dari *layer* di depannya*.* Untuk menghitung seberapa besar gradienatau pengaruh yang diberikan oleh bobot(*weight*) kepada *output layer* adalah dengan menghitung turunan dari fungsi *neuron* terhadap *weight* dengan melibatkan *input* dikalikan dengan *error signal* atau gradien dari *layer* di depannya*.* Untuk menghitung seberapa besar gradienatau pengaruh yang diberikan oleh bias kepada *output layer* adalah dengan menghitung turunan dari fungsi *neuron* terhadap *bias* tanpa melalui proses perkalian yang dimana turunan operasi penjumlahan selalu bernilai 1*.*

Perlu diperhatikan bahwa gradienyang diterima dari *layer* di depannya tersusun dalam bentuk kubus 3 dimensi seperti *feature maps* yang merupakan *output* pada proses *feedforward*. Operasi yang dilakukan untuk menghitung masing masing gradien parameter adalah operasi konvolusi dimana gradien yang diterima atau *dvalues* akan bergeser cell demi cell untuk setiap *kernel* konvolusi ataupun *input* *map*. Pada akhirnya operasi ini akan menghasilkan 3 *output* yang meliputi delta kernel (gradien bobot atau *weight*), delta\_bias (gradien bias), dan delta\_X (gradienfungsi *layer* yang akan dikirimkan pada *layer* di belakangnya).

Gambar 4.20 merupakan potongan kode program *method* atau fungsi dari optimalisasi Rmsprop*.* Potongan kode ini merupakan implementasi dari persamaan 2.22 dan 2.23. Objek kelas optimizer hanya akan melakukan koreksi pada *layer* dengan jenis *fully connected* dan *convolutional* karena kedua jenis *layer* inilah yang memiliki atribut *learnable parameters* atau *kernel* bobot(*weight*), dan *offset* dari fungsi aktivasi yang disebut bias.

|  |
| --- |
| # RMSprop optimizer  class Optimizer\_RMSprop:  def \_\_init\_\_(self, learning\_rate=0.001, decay=0., epsilon=1e-7, rho=0.9):  self.learning\_rate = learning\_rate  self.current\_learning\_rate = learning\_rate  self.decay = decay  self.iterations = 0  self.epsilon = epsilon  self.rho = rho    def pre\_update\_params(self):  if self.decay:  self.current\_learning\_rate = self.learning\_rate \* \  (1. / (1. + self.decay \* self.iterations))  def update\_params(self, *layer*):  if not hasattr(*layer*, 'weight\_cache'):  *layer*.weight\_cache = np.zeros\_like(*layer*.kernel)  *layer*.bias\_cache = np.zeros\_like(*layer*.bias)    *layer*.weight\_cache = self.rho \* *layer*.weight\_cache + \  (1 - self.rho) \* *layer*.delta\_K\*\*2  *layer*.bias\_cache = self.rho \* *layer*.bias\_cache + \  (1 - self.rho) \* *layer*.delta\_b\*\*2  *layer*.kernel += -self.current\_learning\_rate \* \  *layer*.delta\_K / \  (np.sqrt(*layer*.weight\_cache) + self.epsilon)  *layer*.bias += -self.current\_learning\_rate \* \  *layer*.delta\_b / \  (np.sqrt(*layer*.bias\_cache) + self.epsilon)  def post\_update\_params(self):  self.iterations += 1 |

Gambar 4.20 *Code* Algoritma Optimalisasi *RMSProp*

Untuk memudahkan proses komputasi, akan dibuat variabel *cache* yang menyimpan *parameter* *weight* dan *bias* hasil perhitungan *root mean Squere* untuk(t-1). Selanjutnya parameter *weight* dan *bias* akan dikuadratkan sesuai dengan konsep *moving average of the squared gradient (*sebuah mekanisme yang mirip dengan momentum*)* sekaligus melakukan normalisasi menggunakan parameter *rho* seperti yang dituliskan pada persamaan 2.22. Secara default Rmsprop membawa begitu banyak “*momentum-like”* gradien dan pembaharuan *adaptive learning rate*, oleh karenanya parameter rho merupakan *decay factor* untuk menormalisasi atau mengurangi “*momentum-like”* dari gradien (persamaan 2.22) yang terlibat agar model jaringan tidak mengalami *over shoting.* Persamaan 2.23 melakukan *update* atau koreksi pada parameter *weight* dan *bias* menggunakan *learning rate* dan gradien dari *weight* dan *bias*. Konsep *adaptive learning rate* yang dituliskan pada persamaan 2.23berasal dari pembagian *learning* *rate* terhadap akar kuadrat dari gradien (hasil dari persamaan 2.22). Konsep *adaptive learning rate* ini menyebabkan *learning rate* menjadi besar ataupun kecil bergantung pada gradien dari fungsi parameter.

Gambar 4.21 merupakan potongan kode program *method* atau fungsi dari *metode* optimalisasi Adam*.* Potongan kode ini merupakan implementasi dari persamaan 2.16 hingga persamaan 2.21. Objek kelas optimizer hanya akan melakukan koreksi pada *layer* dengan jenis *fully connected* dan *convolutional* karena kedua jenis *layer* inilah yang memiliki atribut *learnable parameters* atau *kernel* bobot(*weight*) dan *offset* dari fungsi aktivasi yang disebut bias.

|  |
| --- |
| def update\_params(self, *layer*):  if not hasattr(*layer*, 'weight\_cache'):  *layer*.weight\_momentums = np.zeros\_like(*layer*.kernel)  *layer*.weight\_cache = np.zeros\_like(*layer*.kernel)  *layer*.bias\_momentums = np.zeros\_like(*layer*.bias)  *layer*.bias\_cache = np.zeros\_like(*layer*.bias)  *layer*.weight\_momentums = self.beta\_1 \* \  *layer*.weight\_momentums + \  (1 - self.beta\_1) \* *layer*.delta\_K  *layer*.bias\_momentums = self.beta\_1 \* \  *layer*.bias\_momentums + \  (1 - self.beta\_1) \* *layer*.delta\_b  weight\_momentums\_corrected = *layer*.weight\_momentums / \  (1 - self.beta\_1 \*\* (self.iterations + 1))  bias\_momentums\_corrected = *layer*.bias\_momentums / \  (1 - self.beta\_1 \*\* (self.iterations + 1))  *layer*.weight\_cache = self.beta\_2 \* *layer*.weight\_cache + \  (1 - self.beta\_2) \* *layer*.delta\_K\*\*2  *layer*.bias\_cache = self.beta\_2 \* *layer*.bias\_cache + \  (1 - self.beta\_2) \* *layer*.delta\_b\*\*2  weight\_cache\_corrected = *layer*.weight\_cache / \  (1 - self.beta\_2 \*\* (self.iterations + 1))  bias\_cache\_corrected = *layer*.bias\_cache / \  (1 - self.beta\_2 \*\* (self.iterations + 1))  *layer*.kernel += -self.current\_learning\_rate \* \  weight\_momentums\_corrected / \  (np.sqrt(weight\_cache\_corrected) + self.epsilon)  *layer*.bias += -self.current\_learning\_rate \* \  bias\_momentums\_corrected / \  (np.sqrt(bias\_cache\_corrected) + self.epsilon) |

Gambar 4.21 *Code* Algoritma Optimalisasi *Adam*

Seperti pada RMSprop, untuk memudahkan proses komputasi, akan dibuat variabel *cache dan momentum* yang menyimpan momen pertama dan momen kedua dari gradien *parameter* *weight* dan *bias.* Adam menggunakan perkiraan momen pertama dan momen kedua untuk menyesuaikan *learning rate,* dimana momen pertama merupakan rata-rata dan momen kedua merupakan *variance* tidak terpusat yang dituliskan dalam dan **.** Dapat dikatakan bahwa *estimator* dan yang dihitung menggunakanpersamaan 2.16 dan 2.18merupakan bentuk penerapan konsep *exponentially moving averages* dalam Adam.

Masing masing *estimator* dan hasil perhitungan dari persamaan 2.16 dan persamaan 2.18 masih mengandung error C atau konstanta dalam momen sehingga selanjutnya dan akan melalui proses koreksi menggunakan persamaan 2.17 dan 2.19. Hasil dari proses koreksi akan disimpan oleh variabel *cache\_corrected* dan *momentum\_corrected. Moving average* hasil dari persamaan 2.17 dan 2.19 akan digunakan untuk *scaling* parameter *learning rate(adaptive learning rate)* pada persamaan 2.20, konsep *adaptive learning rate* ini menyebabkan *learning rate* menjadi besar ataupun kecil bergantung pada gradien dari fungsi parameter. Dilanjutkan dengan proses koreksi parameter *weight* dan *bias* menggunakan persamaan 2.21.

## Implementasi Tahap *Testing*

|  |
| --- |
| mylenet.load\_parameters(mainPath=mainPath,*epoch*s=*epoch*s,method=method,)  mylenet.lenet\_predictions(X\_test, Y\_test)  imgpath= "C:/Users/ASUS/Documents/py/cnn/path/to/image.jpg "  temp = os.path.split(imgpath)  prob = mylenet.one\_image(mylenet.*layer*s, imgpath )  print("image test::",imgpath)  print("\nFile Name ::", temp[1], " Tipe kain ::", data.labelName[np.argmax(prob)], "||" ,"confidence ::", prob[0,np.argmax(prob)])  def feedForward(X, *layer*s, y\_*true*):  inp = X  wsum = 0  y\_*true* = y\_*true*  for *layer* in *layer*s:  if isinstance(*layer*, FC\_*LAYER*) and len(inp.shape) == 4:  inp, ws = *layer*.forward(inp.reshape(inp.shape[0], inp.shape[1]\*inp.shape[2]\*inp.shape[3]))  elif isinstance(*layer*, Activation\_Softmax\_Loss\_CategoricalCrossentropy):  loss, ws = *layer*.forward(inp,y\_*true*)  inp = *layer*.*output*  else:  inp, ws = *layer*.forward(inp)  wsum += ws  return inp, loss, wsum  def lenet\_predictions(self, X, Y):  predictions, loss ,weight\_sum = LENET5.feedForward(X, self.*layer*s, Y)  y\_*true* = np.argmax(Y, axis=1)  y\_pred = np.argmax(predictions, axis=1)  print("Dataset accuracy: ", accuracy\_score(y\_*true*, y\_pred)\*100)  print("FeedForward time:", stop - start)  tn, fp, fn, tp = confusion\_matrikss(y\_true, y\_pred)  confusion = confusion\_matrikss(y\_true, y\_pred, target\_names=['Bali', 'Cendana', 'Indonesia pink', 'Tri color Sudamala']))  print('Confusion Matrikss\n')  print(confusion) |

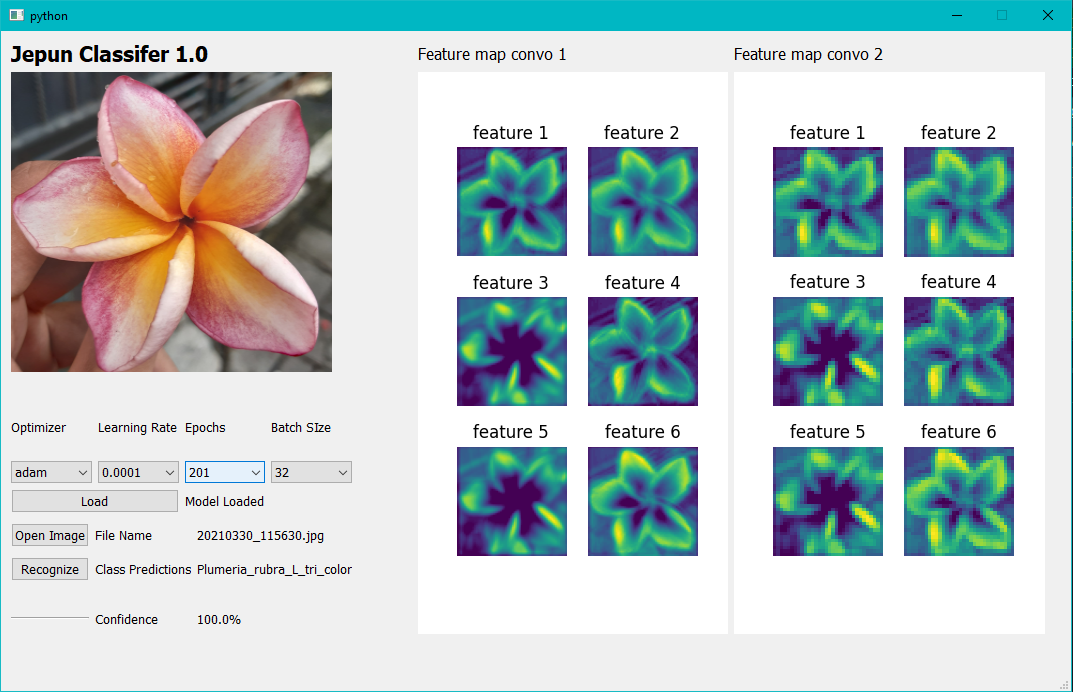
Gambar 4.22 *Code* Tahap *Testing*

Gambar 4.22 merupakan potongan Code yang menghandle proses *testing* dengan menggunakan dataset *testing*. Proses *testing* hanya meliputi tahap *feedforward* dari model CNN yang terbentuk, dimana seluruh citra *testing* akan diproses secara sekaligus. Setelah model mengembalikan suatu *output* yang berupa y\_pred atau data yang meliputi sekumpulan kelas prediksi, data prediksi akan dihitung dan dicocokan dengan kelas asli, lalu performa global atau akurasi akan dihitung menggunakan persamaan 2.45. Selanjutnya program akan menghitung nilai precision, recall dan F1-measure berdasarkan persamaan 2.46, 2.47, 2.48 menggunakan y\_pred dan y\_true dari model.

## Implementasi Sistem

|  |
| --- |
| def one\_image(*layer*s, path):  inp = cv2.imread(path)  inp = cv2.cvtColor(inp, cv2.COLOR\_BGR2RGB)  if inp.shape[1] > 64 and inp.shape[0] > 64:  dim = (64, 64)  inp = cv2.resize(inp, dim, interpolation = cv2.INTER\_AREA)  inp = rearrange(inp, ' h w c -> c h w ')  for *layer* in *layer*s:  if isinstance(*layer*,CONV\_*LAYER*) and len(inp.shape) == 3:  inp, ws = *layer*.forward(inp.reshape(1,inp.shape[0],inp.shape[1],inp.shape[2]))  elif isinstance(*layer*, FC\_*LAYER*) and len(inp.shape) == 4:  inp, ws = *layer*.forward(inp.reshape(inp.shape[0], inp.shape[1]\*inp.shape[2]\*inp.shape[3]))  elif isinstance(*layer*, Activation\_Softmax\_Loss\_CategoricalCrossentropy):  *output* = *layer*.guessing(inp)  else:  inp, ws = *layer*.forward(inp)  #print("prob",inp)  return *output* |

Gambar 4.23 *Code* Proses Pengenalan Oleh Sistem



Gambar 4.24 *User* *Interface* Dari Sistem

Gambar 4.37 merupakan implementasi *User* *Interface* atau UI dari sistem klasifikasi bunga jepun. Pada UI terdapat 4 *combo box* dan 4 tombol yang dapat digunakan oleh *user* untuk berinteraksi dengan sistem. 4 *combo box* pada sisi kiri UI berfungsi untuk memberikan keleluasaan kepada user dalam memilih model CNN berdasarkan *tuning parameter* pembentuk yang dikehendaki oleh *user.* Setelah memilih *tuning parameter*, pastikan *user* melakukan aksi klik pada tombol *load* untuk memuat model yang tersimpan. Tombol *open image* digunakan untuk memilih foto bunga jepun atau *Plumeria sp.* yang akan dikenali oleh user. Setelah gambar dipilih dan dimuat oleh sistem, gambar akan ditampilkan pada suatu *image space* di dalamUIseperti pada gambar 4.37. Tombol *recognize* berfungsi untuk melakukan proses pengenalan dari citra yang sudah dimuat oleh sistem. Setelah melakukan proses pengenalan, *output* kelas dan tingkat kepercayaan dari kelas yang dihasilkan oleh sistem. Sistem juga akan menampilkan *feature map* dari citra *input* yang dibaca oleh kedua layer konvolusi dari model CNN pada dua canvas di dalam UI.

Gambar 4.36 merupakan potongan kode yang digunakan untuk melakukan proses pengenalan untuk satu citra *input*. Kode ini bersifat fleksibel terhadap citra *input*, jika citra *input* memiliki dimensi yang lebih besar dari 64 x 64 px, maka sistem akan melakukan *scaling* untuk mengecilkan resolusi citra. *Input* citra selanjutnya melalui proses *feedforward* hingga pada *layer* klasifikasi dan aktivasi *softmax.* Setelahproses *feedforward* pada *layer* aktivasi *softmax*, sistem akan memberikan *output* berupa probabilitas untuk masing masing kelas prediksi, *output* ini merupakan *output* dari *layer* aktivasi *softmax*. Selanjutnya masing masing probabilitas akan dicocokan pada label kelas dari sistem sehingga ditemukan kelas dengan tingkat probabilitas prediksi terbesar*.*