Klasifikasi Bunga dengan Multilayer Perceptron

Chrystian 18/430257/PA/18770
Karunia Eka Putri 18/430265/PA/18778
Widad Abida Rahmah 18/430275/PA/18788

May 26, 2021

1 Arsitektur

Multilayer Perceptron adalah topologi di mana perceptron perceptron terhubung membentuk beberapa lapisan (layer). Sebuah MLP mempunyai lapisan masukan (input layer), minimal satu lapisan tersembunyi (hidden layer), dan lapisan luaran (output layer). (Negnevitsky, Michael, 2005).

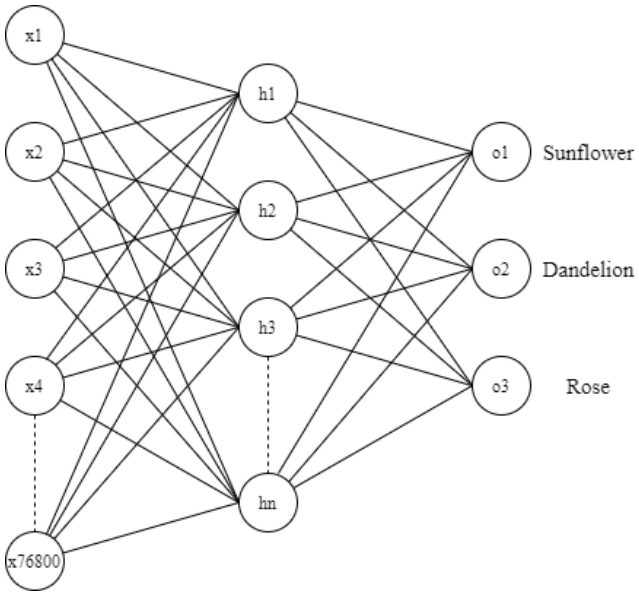

Figure 1: **Arsitektur dari Multilayer Perceptron model**. Terdapat $x_{(240,320)} \rightarrow x_{(76800)}$ Input layer dengan 3 Output layer yang merepresentasikan bunga *Sunflower, Dandelion,* dan *Rose*. Terdapat 1 hidden layer dengan hasil eksperimen terbaik ketika jumlah neuron $h_n = 16$ dengan $lr = 0.1$ (Section 12).

Arsitektur ini memiliki 3 layer, yaitu layer input, 1 hidden layer, dan layer output. Node masukan berjumlah 76800 (240x320) dan node keluaran berjumlah 3 (sunflower, dandelion, dan rose) sedangkan untuk node pada hidden layer akan dilakukan eksperimen untuk mencari jumlah node dengan hasil terbaik.

2 Implementasi Arsitektur

Membuat model dilakukan dengan menentukan banyak neuron pada setiap layer, fungsi error, dan fungsi aktivasi yang ingin digunakan. Kode di bawah ini digunakan untuk membuat model dengan fungsi aktivasi sigmoid dan fungsi lossnya binary cross entropy loss.

```
def create_model(hidden_layer_n):
      dense_0 = Dense(76800, hidden_layer_n)
      activation0 = ActivationSigmoid()
      dense_1 = Dense(hidden_layer_n, 3)
      activation1 = ActivationSigmoid()
      CELoss = BinaryCrossEntropyLoss()
      return [dense_0, activation0, dense_1, activation1, CELoss]
```

Listing 1: Membuat Model

Fungsi create_model menerima input berupa banyak neuron pada hidden layer. Sementara input dan output layer sudah ditentukan yaitu, input layer sebanyak 76800 neuron (320 x 240 pixel) , output layer sebanyak tiga neuron (sunflower, dandelion, rose).

3 Implementasi Load Dataset dan Visualisasi Data

3.1 Load Dataset

```
labels = ['Sunflower', 'Dandelion', 'Rose']
  n_data = 100

  X = None
  y = None
  for i in range(len(labels)):
      if(X is None):
          X = getImages(labels[i], n=n_data)
      else:
          X = np.append(X, getImages(labels[i], n=n_data), axis=0)

      if(y is None):
          y = np.full(shape=n_data,fill_value=i,dtype=np.int)
      else:
          y = np.append(y, np.full(shape=n_data,fill_value=i,dtype=np.int), axis=0)

  print(X.shape)
  X = convertToGrayscale(X)
  print(X.shape)
  X = preprocessImage(X)
  print(X.shape)
```

Listing 2: Kode Python untuk memuat data

Implementasi dari Load Dataset, pertama kami mendefinisikan labels berisi list dari nama bunga yang akan diklasifikasi. Untuk Ilkom B, labels yang akan dilakukan klasifikasi adalah Sunflower, Dandelion, dan Rose. Setelah mendefinisikan label kami mendefinisikan n_data untuk mensample sebanyak n_data untuk setiap label, dalam kasus ini 100 untuk setiap label.

Untuk memuat data gamber kami membuat fungsi getImages yang akan dijelaskan pada figur berikutnya. Kami mendefinisikan X untuk menyimpan data-data semua gambar yang dimuat. Kami menggunakan np.append(X, new_data, axis=0) untuk menambahkan data baru pada axis 0. Setiap label memiliki 100 data, 240 height, 320 width, 3 RGB dengan melakukan operasi append maka dimensi dari X menjadi

$$(100, 240, 320, 3) + (100, 240, 320, 3) + (100, 240, 320, 3) \rightarrow (300, 240, 320, 3)$$

Setelah dimuat, X data dilakukan convertToGrayscale melakukan operasi pada 3 RGB channel membuat

$$(300, 240, 320, 3) \rightarrow (300, 240, 320)$$

dan Preprocessing melakukan normalisasi dan menggabungkan dimensi height dan width membuat dimensi akhir X menjadi

$$(300, 240, 320) \rightarrow (300, 76800)$$

(section 4).

Untuk y label sama seperti X untuk setiap label akan dilakukan operasi append pada axis 0. Setiap label akan dibuat sebanyak n_data dengan nilai i. Kami menggunakan fungsi numpy np.full(shape=n_data, fill_value=i, dtype=np.int).

```
def getImages(label, n=100):
    """return list of (default 100) images for a given label"""

    # Check Dataset
    checkDataset()

    # Try and get path
    path = "data/flowers/" + label.lower()
    if(not os.path.exists(path)):
        print("No label found")
        return -1
```

```
    # Get all images address
    allFiles = fnmatch.filter(os.listdir(path), '*.jpg')

    usedFiles = random.sample(allFiles, n) # get n samples from allFiles

    # Read the data into numpy array
    imagesRGB = []
    for file in usedFiles:
        img = Image.open(path+'/'+file)
        img = img.resize((320,240), Image.ANTIALIAS)
        imagesRGB.append(np.asarray(img))

    imagesRGB = np.asarray(imagesRGB) #(n, 240, 320, 3)
```

Listing 3: Fungsi getImages untuk mengambil dan sample n banyak data dari suatu label

Fungsi getImages memakai 2 parameter yaitu nama bunga 'label' dan 'n' yaitu banyak sample data yang akan diambil dari dataset. Langkah pertama kami mengecek dataset dengan checkDataset(), apabila tidak maka dengan Kaggle API data akan mendownload secara otomatis.

Langkah selanjutnya dengan path, flowers dataset dapat dimuat. Apabila bunga Sunflower ingin dimuat, maka kami dapat mengambil gambar dari path 'flowers/sunflower/*'. Dengan ini dengan menggabungkan string path root dataset dengan label.lower() dapat memuat gambar yang diinginkan. Dengan fnmatch kami dapat mecari seluruh file bertipe gambar atau format *.jpg yang ada pada suatu directory (yang didapatkan dengan os.listdir(path)). Seluruh gambar allFiles akhirnya dapat disample sebanyak n dengan fungsi random.sample(allFiles, n).

Langkah akhir, untuk setiap data yang ada pada usedFiles akan dimuat dengan library python Images. Pertama gambar akan dibuka dengan Image.open, kedua gambar akan di resize menjadi (320, 240) dengan mode antialias, dan terakhir image yang telah diresize akan di append kepada list imagesRGB. Semua data disimpan sebagai numpy array untuk mempermudah operasi matematik dan fungsi akan mengembalikan data imagesRGB yang berisi n banyak gambar data label.

3.2 Visualisasi Data

```
import matplotlib.pyplot as plt
def showImages(images, label, cmap='viridis'):
    fig, axs = plt.subplots(2, 2, figsize=(8,6))
    fig.suptitle(label+" images from dataset.", fontsize=16)

    axs[0, 0].imshow(images[0], cmap=cmap)
    axs[0, 1].imshow(images[1], cmap=cmap)
    axs[1, 0].imshow(images[2], cmap=cmap)
    axs[1, 1].imshow(images[3], cmap=cmap)
```

Listing 4: Fungsi showImages untuk memvisualisasi 4 data gambar teratas

Untuk proses visualisasi data kami menggunakan library Matplotlib.pyplot, dan menggunakan subplot untuk menvisualisasi 4 gambar sekaligus. Fungsi ini memakai parameter 'images' yang berisi data gambar-gambar, 'label' yang berisi nama, dan 'cmap' untuk memberi informasi color map dari data yang digunakan.

```
labels = ['Sunflower', 'Dandelion', 'Rose']

for label in labels:
    Images = getImages(label)
    showImages(Images, label)

    ImagesGrayscale = convertToGrayscale(Images)
    showImages(ImagesGrayscale, label+" Grayscale", cmap='gray')
```

Listing 5: Penggunaan showImages

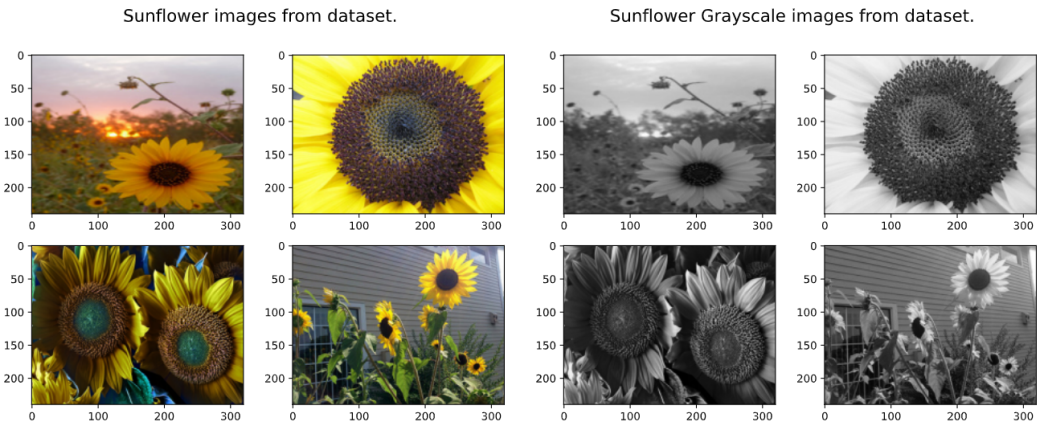

Figure 2: Hasil Visualisasi Sunflower

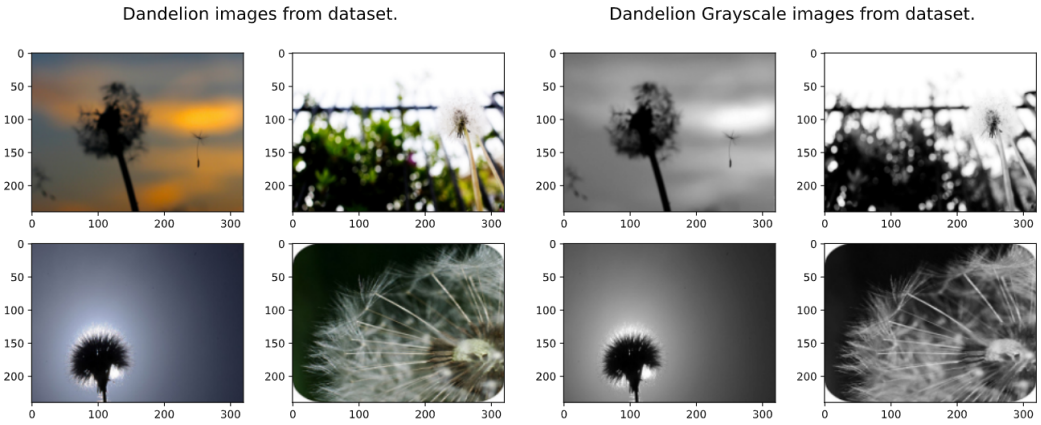

Figure 3: Hasil Visualisasi Dandelion

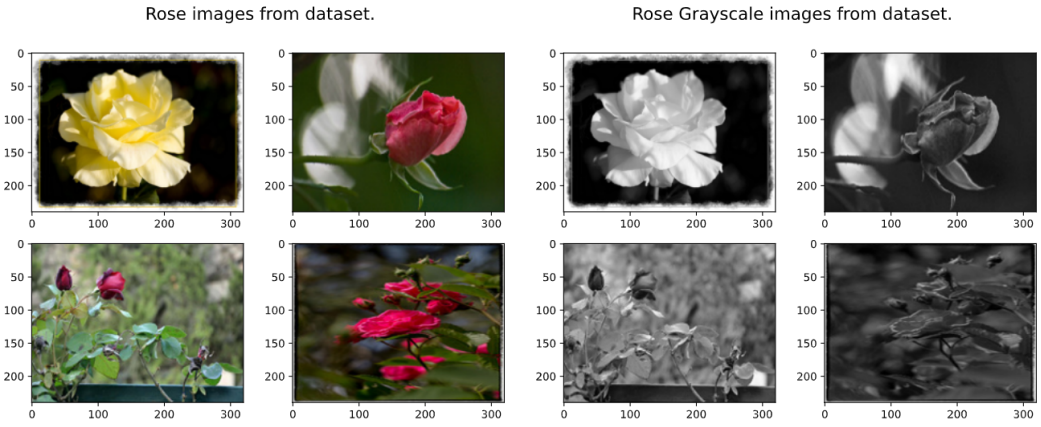

Figure 4: Hasil Visualisasi Roses

4 Implementasi Fungsi Konversi Grayscale dan Preprocessing

4.1 Fungsi Konversi Grayscale

```
def convertToGrayscale(imageRGB):
    """return grayscale conversion of RGB image"""
    # https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html
```

```
        # Grayscale = 0.299 * R + 0.587 * G + 0.114 * B
        # We could easily vectorized and achieve this by dot product of imageRGB with
[0.299,0.587,0.114]

        imageGrayscale = np.dot(imageRGB, [0.299,0.587,0.114])

        return imageGrayscale
```

Listing 6: Fungsi Konversi Grayscale

$$\vec{X}_{grayscale} = \vec{X} \cdot \begin{bmatrix} .299 \\ .587 \\ .114 \end{bmatrix}$$

$$(n, 240, 320, 3) \cdot (3) \rightarrow (n, 240, 320)$$

4.2 Fungsi Preprocessing

```
    def preprocessImage(images):
        """return np.array to be loaded into a model,
        flatten dimension (100, 240, 320) -> (100, 76800), and
        Normalize data value from int [0, 255] -> float [0,1]"""

        x = images.reshape(*images.shape[:1], -1) #(100, 76800)
        x = x/255.0
        return x
```

Listing 7: Fungsi Preprocessing

Pada fungsi preprocessing terjadi 2 operasi yang akan dilakukan yaitu flatten dimensi width dan height menjadi satu, operasi ini dapat dilakukan dengan np.reshape(*images.shape[:1],-1), dan setelah itu normalisasi dari tipe integer menjadi float dengan $\vec{X} = \vec{X}/255$..

5 Implementasi Aktivasi

5.1 Aktivasi Sigmoid

Fungsi ini digunakan untuk memetakan nilai output menjadi nilai o atau 1. Formula aktivasi sigmoid :

$$g(x) = \frac{1}{1 + e^{-x}}$$

```
    class ActivationSigmoid:
        def forward(self, inputs):
            self.inputs = inputs
            self.output = 1/ (1+np.exp(-inputs))

        def backward(self, dvalues):
            self.dinputs = dvalues * (1-self.ouput) * self.output

        def predictions(self, outputs):
            return (outputs> 0.5) * 1
```

Listing 8: Aktivasi Sigmoid

5.2 Aktivasi Softmax

Fungsi aktivasi softmax digunakan pada output layer untuk menentukan probabilitas. Probabilitas dihitung dengan cara membagikan hasil eksponen dari output neuron ke-k hidden layer (I_k) dengan penjumlahan hasil eksponen dari output semua neuron hidden layer. Formula aktivasi softmax:

$$y_k = \frac{e^{I_k}}{\sum_\alpha e^{I_\alpha}}, 1 \leq k \leq \alpha$$

Pada kode di bawah ini semua nilai eksponen dari output neuron disimpan dalam variabel exp_values untuk kemudian dihitung probabilitasnya dengan rumus exp_values dibagi jumlah (sum) dari semua nilai eksponen dari output neuron.

```
class ActivationSoftmax:
    def forward(self, inputs):
        self.inputs = inputs
        exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
        probabilities = exp_values / np.sum(exp_values, axis=1, keepdims=True)
        self.output = probabilities
```

Listing 9: Aktivasi Softmax

Karena arsitektur yang digunakan adalah multi layer perceptron (MLP) sementara aktivasi softmax digunakan hanya pada output layer sehingga diperlukan fungsi aktivasi lain untuk digunakan pada hidden layer. Salah satu fungsi aktivasi yang bisa digunakan adalah aktivasi ReLU (Rectified Linear Unit. Secara matematis fungsi ReLU dapat didefinisikan sebagai:

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

or

$$f(x) = max(0, x)$$

Pada class ActivationRelu di bawah ini ada dua fungsi, yaitu forward dan backward. Fungsi forward digunakan saat forward pass untuk menghitung nilai max(0, input). Sedangkan fungsi backward digunakan saat backward pass untuk menghitung gradient yang nanti akan digunakan untuk update bobot dan bias.

```
class ActivationReLU:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0
```

Listing 10: Aktivasi ReLU

6 Implementasi Inisialisasi Bobot dan Bias Dense Layer

Dalam class Dense pada kode di bawah ini, bobot diinisialisasi dengan bilangan random sedangkan bias diinisialisasi dengan bilangan 0.

```
class Dense:
    def __init__(self, n_inputs, n_neurons):
        self.W = np.random.rand(n_inputs, n_neurons)/100.0
        self.b = np.zeros((1, n_neurons))

    def forward(self, inputs):
        ...

    def backward(self, dvalues):
        ...
```

Listing 11: Inisialisasi Bobot dan Bias

Bobot dan bias direpresentasikan dalam bentuk matriks berukuran m x n dan 1 x n. Dengan m = n_inputs dan n = n_neurons, bentuk matriks bobot:

$$\vec{W} = \begin{bmatrix} \theta_{11} & \theta_{12} & ... & \theta_{1n} \\ \theta_{21} & \theta_{22} & ... & \theta_{2n} \\ ... & ... & ... & ... \\ \theta_{m1} & \theta_{m2} & ... & \theta_{mn} \end{bmatrix} = \begin{bmatrix} rand() & rand() & ... & rand() \\ rand() & rand() & ... & rand() \\ ... & ... & ... & ... \\ rand() & rand() & ... & rand() \end{bmatrix}$$

dan matriks bias:

$$\vec{b} = \begin{bmatrix} b_1 & b_2 & ... & b_n \end{bmatrix} = \begin{bmatrix} 0. & 0. & ... & 0. \end{bmatrix}$$

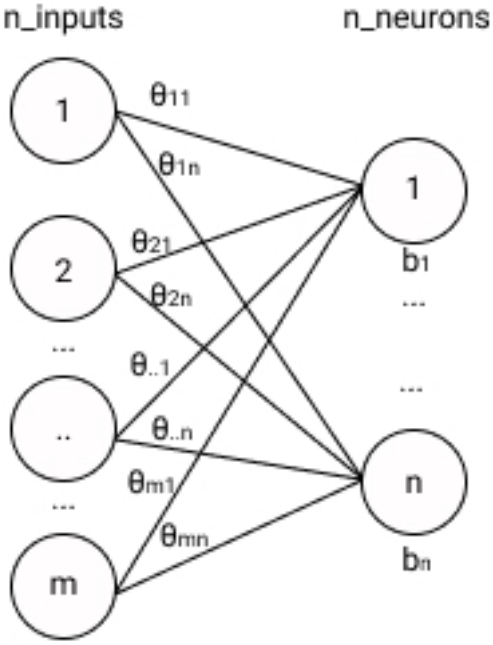

Figure 5: Dense

7 Implementasi Fungsi Error

7.1 Binary Cross Entropy Loss

Formula Binary Cross Entropy Loss :

$$H(\hat{y}) = -\frac{1}{N}\sum_{i}^{N} y_i log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

Dengan $\hat{y}_i$ adalah output probability (y_predict), y_i adalah label (y_true), formula diatas apabila diimplementasikan ke dalam kode Python akan menjadi seperti ini:

```
    class BinaryCrossEntropyLoss:
          def forward(self, y_pred, y_true):
              y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
              sample_losses = -(y_true * np.log(y_pred_clipped) + (1 - y_true) * np.log(1 -
 y_pred_clipped))
              sample_losses = np.mean(sample_losses, axis=-1)

              # Return losses
              return sample_losses
```

Listing 12: Fungsi Binary Cross Entropy

Pada kode di atas loss akan dihitung satu per satu untuk setiap input data, baru kemudian dijumlahkan dan dibagi dengan banyak data dengan numpy mean.

7.2 Categorical Cross Entropy Loss

Untuk Loss function karena kami menggunakan Softmax sejak terdapat 3 kelas untuk klasifikasi, kami memakai Categorical Cross Entropy.

$$E = -\sum y_i \log \hat{y}_i$$

$$\hat{y}_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

Dengan $\hat{y}_i$ adalah output probability, x output neuron sebelum, dan y_i adalah label.

```
class CategoricalCrossEntropyLoss:
    def forward(self, y_pred, y_true):
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7) #prevent log(0)
        correct_confidences = np.sum(y_pred_clipped * y_true, axis=1)

        # Losses
        CEntropy = -np.log(correct_confidences)
        return CEntropy
```

Listing 13: fungsi Categorical Cross Entropy

Variabel y_pred adalah hasil output dari layer softmax sebelumnya, dan y_true adalah label. Sebelum dilakukan operasi log, y_pred di batasi antara 1e-7 sampai 1- 1e-7 untuk mencegah error operasi $\log 0$. Karena y_i one hot encoded maka dapat dikalikan langsung dengan y_pred dan dapat dijumlah pada axis 1. setelah itu Categorical Cross Entropy dapat dilakukan dengan fungsi numpy -np.log(correct_confidence).

8 Implementasi Feedforward

Dalam proses feedforward input akan dijadikan bentuk linear:

$$h(x; \theta, b) = \theta_1 x_1 + \theta_2 \cdot x_2 + ... + \theta_m \cdot x_m + b$$

Karena linearisasi tersebut berlaku untuk semua x dalam matriks $\vec{X}$ maka dapat ditulis menjadi:

$$\vec{I} = \vec{X} \cdot \vec{W} + \vec{b}$$

$$\vec{I} = \begin{bmatrix} W_{11} & W_{12} & ... & W_{1n} & b_1 \\ W_{21} & W_{22} & ... & W_{2n} & b_2 \\ ... & ... & ... & ... & ... \\ W_{m1} & W_{m2} & ... & W_{mn} & b_m \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ ... \\ X_m \\ 1 \end{bmatrix}$$

Apabila dituliskan dalam kode dengan library numpy akan menjadi sebagai barikut:

```
class Dense:
        ...
    def forward(self, inputs):
        self.inputs = inputs
        self.output = np.dot(self.inputs, self.W) + self.b
        ...
```

Listing 14: Feedforward Dense

Yang terjadi adalah input masuk lalu dikalikan dengan weight dan ditambah dengan bobot lalu dimasukkan ke fungsi aktivasi layer selanjutnya. Hal ini terjadi berulang-ulang di tiap layer hingga sampai di output layer.

9 Implementasi Backpropagation Backward

9.1 Sigmoid Activation dan Binary Cross Entropy Loss

Derivative dari fungsi binary cross entropy loss adalah sebagai berikut:

$$\frac{dH}{d\hat{y}_i} = -1(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i})$$

Jika digunakan aktivasi sigmoid dan binary cross entropy loss untuk backward pada output layer maka implementasi dalam kode akan menjadi seperti ini:

```
    class BinaryCrossEntropyLoss:
            ...

        def backward(self, dvalues, y_true):
            samples = len(dvalues)
            outputs = len(dvalues[0])
            clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)

            self.dinputs = -(y_true / clipped_dvalues - (1 - y_true) / (1 -
clipped_dvalues)) / outputs
            self.dinputs = self.dinputs / samples
```

Listing 15: Categorical Cross Entropy Loss

9.2 Softmax Activation dan Categorical Cross Entropy Loss

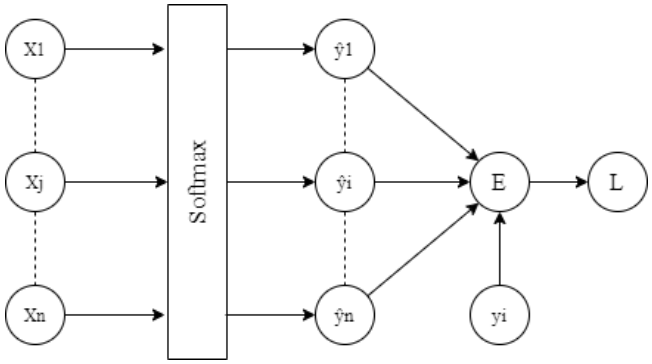

Figure 6: Softmax and Cross Entropy Loss Architecture

Pada output layer digunakan fungsi Softmax dan Categorical Cross Entropy Loss

$$E = -\sum y_i \log \hat{y}_i$$

$$\hat{y}_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

Kemudian dengan menggunakan backpropagation dan chain rule, persamaan backward dari layer Softmax dan Loss dapat di tuliskan dengan,

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial x_j}$$

Untuk *partial derivative* dari output probability $\hat{y}_i$ terhadap input x_i,

$$i = j, \frac{\partial \hat{y}_i}{\partial x_i} = \frac{\sum_k e^{x_k} \frac{\partial e^{x_i}}{\partial x_i} - e^{x_i} \frac{\partial \sum_k e^{x_k}}{\partial x_i}}{(\sum_k e^{x_k})^2} = \frac{e^{x_i}(\sum_k e^{x_k} - e^{x_i})}{(\sum_k e^{x_k})^2} = \hat{y}_i(1 - \hat{y}_i)$$

$$i \neq j, \frac{\partial \hat{y}_i}{\partial x_j} = \frac{\sum_k e^{x_k} 0 - e^{x_i} \frac{\partial \sum_k e^{x_k}}{\partial x_j}}{(\sum_k e^{x_k})^2} = \frac{-(e^{x_i})e^{x_j}}{(\sum_k e^{x_k})^2} = -\hat{y}_i \hat{y}_j$$

Untuk backward Loss Function, *partial derivative* dari E terhadap $\hat{y}_i$,

$$\frac{\partial E}{\partial \hat{y}_i} = -\sum y_i \frac{\partial \log \hat{y}_i}{\partial \hat{y}_i} = -\sum y_i \frac{1}{\hat{y}_i}$$

Dengan ini dapat disimpulkan,

$$\frac{\partial E}{\partial x_j} = -\sum_{i \neq j} y_i \frac{1}{\hat{y}_i}(-\hat{y}_i \hat{y}_j) - \sum_{i=j} yi \frac{1}{\hat{y}_i} \hat{y}_i(1 - \hat{y}_i)$$

$$\frac{\partial E}{\partial x_j} = \sum_{i \neq j} y_i(\hat{y}_j) - y_j(1 - \hat{y}_j) = \sum_{i \neq j} y_i \hat{y}_j + y_j \hat{y}_j - y_j = \sum_i y_i \hat{y}_j - y_j$$

Karena label y_i one hot encoded, maka didapatkan:

$$\frac{\partial E}{\partial x_j} = \hat{y}_j - y_j$$

Dari pembahasan di atas dapat apabila diimplementasikan dalam bentuk kode maka akan menjadi seperti ini:

```
class CategoricalCrossEntropyLoss:
        ...

    def backward(self, dvalues, y_true):
        samples = len(dvalues)

        y_true = np.argmax(y_true, axis=1)

        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Listing 16: Categorical Cross Entropy Loss

Fungsi backward menerima parameter inputs berupa output dari hidden layer yang telah dikenai fungsi softmax (y_prediksi), lalu dihitung gradient dengan cara mengurangkan y_prediksi dengan y_target secara sekaligus. Karena menggunakan hot encoding, maka cukup satu dari tiga inputs (neuron) yang dikurangi dengan nilai satu.

```
        self.dinputs[range(samples), y_true] -= 1
```

Hasil penghitungan gradient ini akan disimpan dalam variabel dinputs, yang selanjutnya akan di backward pass oleh fungsi backward yang lain. Fungsi backward ini digunakan untuk backward dari output layer ke hidden layer saja. Fungsi backward dari hidden layer ke output layer menggunakan fungsi backward pada class-class fungsi aktivasi yang telah dibahas pada poin sebelum-sebelumnya.(Lihat 5.1 dan 5.3)

9.3 Dense Layer

Pada class Dense selain terdapat fungsi forward dan constructor juga terdapat fungsi backward untuk backpropagation.

```
class Dense:
    def __init__(self, n_inputs, n_neurons):
        ...

    def forward(self, inputs):
        ...

    def backward(self, dvalues):
        self.dW = np.dot(self.inputs.T, dvalues)
        self.db = np.sum(dvalues, axis=0, keepdims=True)

        self.dinputs= np.dot(dvalues, self.W.T)
```

Listing 17: Dense

Fungsi backward menerima parameter berupa nilai δ atau gradient dari layer sebelumnya (dvalues), lalu akan dicari gradient dari bobot (dW) dan bias (db) pada layer tersebut. Gradient atau perubahan bobot dicari dengan mengalikan nilai inputs pada layer tersebut (yang didapat saat proses forward pass) dengan nilai dvalues. Sedangkan gradient atau perubahan bias dicari dengan menjumlahkan semua nilai dvalues sehingga menghasilkan nilai sebanyak neuron pada sebelumnya. Kemudian akan dicari dinputs yang mana nanti akan digunakan untuk backward selanjutnya.

9.4 Backpropagation Update Weight

Formula untuk update bobot adalah:

$$\theta_{baru} = \theta_{lama} - \alpha \cdot \Delta\theta$$

Formula untuk update bias adalah:

$$b_{baru} = b_{lama} - \alpha \cdot \Delta b$$

dengan α = learning rate, $\Delta\theta$ = perubahan bobot (ΔW), Δb = perubahan bias. Apabila diimplementasikan dalam kode akan menjadi seperti ini :

```
    class Backpropagation:
          def __init__(self, lr):
              self.lr = lr

          def updateParams(self, layer):
              layer.W += -self.lr * layer.dW
              layer.b += -self.lr * layer.db
```

Listing 18: Update Bobot

Fungsi updateParams akan meng-*update* bobot dan bias pada setiap layer dengan mengurangkan bobot atau bias lama dengan perubahan bobot yang telah dikalikan dengan learning rate.

10 Implementasi Fungsi Prediksi

Prediksi apabila dituliskan dalam kode menjadi:

```
      def prediction(y_prob):
          return np.argmax(y_prob, axis=-1)

    def predict(X):
          dense_0.forward(X)
          activation0.forward(dense_0.output)
          dense_1.forward(activation0.output)
          activation1.forward(dense_1.output)

          return prediction(activation1.output)
```

Listing 19: Fungsi Prediksi

Fungsi prediksi adalah penerapan lanjutan dari feedforward dimana input akan diolah (dengan feedforward) dan dibawa ke hidden layer lalu diolah lagi (dengan feedforward) dan dibawa ke output layer hingga menghasilkan nilai prediksi.

11 Implementasi Fungsi Akurasi

Data yang sudah dilakukan pengolahan data kemudian diklasifikasikan berdasarkan kondisi yang sebenarnya dan hasil prediksi. Data tersebut lalu dihitung tingkat akurasinya menggunakan rumus sehingga didapatkan nilai prosentase tingkat akurasinya.

Formula untuk akurasi adalah:

$$Akurasi = \frac{totalprediksibenar}{totalprediksi} \times 100$$

Apabila diimplementasikan dalam kode akan menjadi:

```
def accuracy(y_pred, y_true):
  return (np.argmax(y_pred, axis=-1) == np.argmax(y_true, axis=-1)).mean()
```

Listing 20: Fungsi Akurasi

Dimana apabila value prediksi sama dengan value yang sebenarnya akan menghasilkan angka 1 sedangkan jika berbeda akan menghasilkan angka 0. Kemudian angka 1 dan 0 dijumlahkan dan dirata-rata untuk menghasilkan nilai akurasi.

12 Implementasi Fungsi Training dan Testing

Implementasi kode untuk membagi data menjadi data training 80% and data testing 20%:

```
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state
=42, stratify=y)
```

Listing 21: Split Data

Implementasi fungsi training:

```
  def fit(x_train, y_train, lr):
        backprop = Backpropagation(lr)
        epochs = 50
        loss = []
        acc = []

        n_data = len(x_train)
        n_batch = n_data//batch_size

        remainder = n_data%batch_size
        if(remainder!=0):
            n_batch += 1

        for _ in tqdm(range(epochs), leave=False):
            batch_loss = []
            batch_acc = []

            for i in tqdm(range(n_batch), leave=False):
                #if batch can not be equally divided
                #batch all of the remainder data
                if(i*batch_size+batch_size>n_data):
                    x_batch = x_train[i*batch_size:]
                    y_batch = y_train[i*batch_size:]
                else:
                    x_batch = x_train[i*batch_size: (i+1) *batch_size]
                    y_batch = y_train[i*batch_size: (i+1) *batch_size]

                #Forward pass
                dense_0.forward(x_batch)
                activation0.forward(dense_0.output)
                dense_1.forward(activation0.output)
                activation1.forward(dense_1.output)

                #Backward pass
                CELoss.backward(activation1.output, y_batch)
                dense_1.backward(CELoss.dinputs)
                activation0.backward(dense_1.dinputs)
                dense_0.backward(activation0.dinputs)

                #Update Weight
                backprop.updateParams(dense_1)
                backprop.updateParams(dense_0)

                #Calculate Metrics/ batch
                batch_acc.append(accuracy(activation1.output, y_batch))
                batch_loss.append(CELoss.forward(activation1.output, y_batch))

            acc.append(np.mean(batch_acc))
            loss.append((np.sum(batch_loss[:n_batch-1])+np.sum(batch_loss[n_batch-1]))/
n_data)
        return acc, loss
```

Listing 22: Fungsi Training

Visualisasi fungsi training (forward pass and backward pass) dalam gambar:

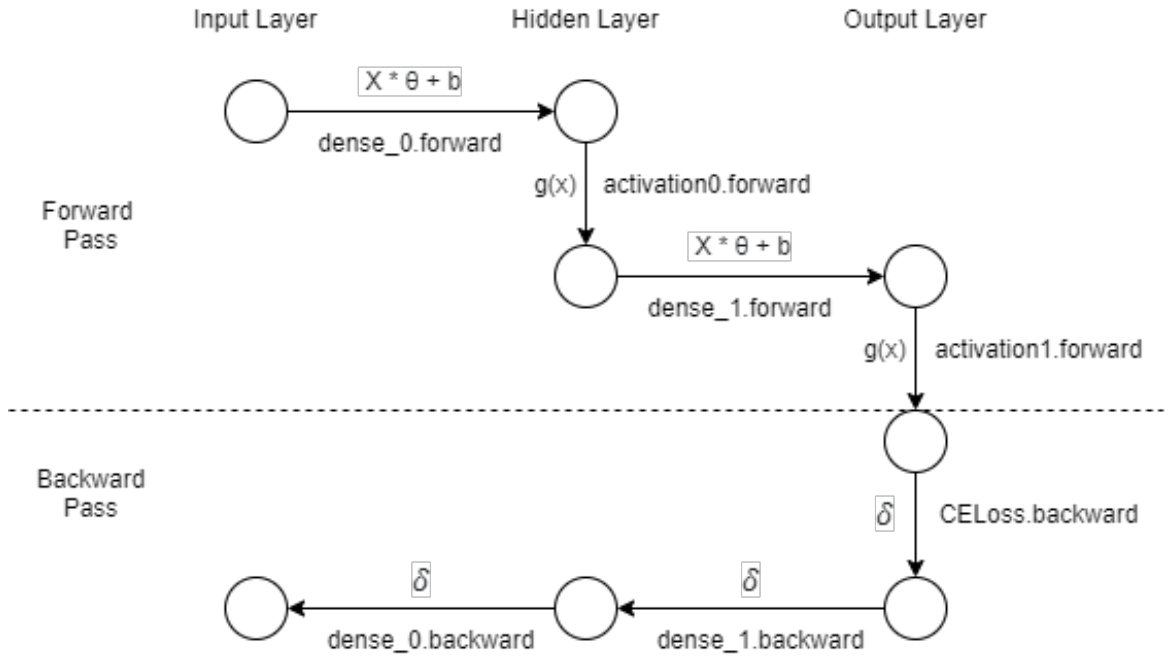

Figure 7: Fungsi Training

Implementasi fungsi testing:

```
def test(x_test, y_test):
      dense_0.forward(X_test)
      activation0.forward(dense_0.output)
      dense_1.forward(activation0.output)
      activation1.forward(dense_1.output)

      acc = accuracy(activation1.output, y_test)
      loss = np.mean(CELoss.forward(activation1.output, y_test))
      return acc, loss
```

Listing 23: Fungsi Testing

Penggunaan fungsi training:

```
# Neuron Hidden Layer = 16
[dense_0, activation0, dense_1, activation1, CELoss] = create_model(16)
acc_16, loss_16 = fit(X_train, y_train, 0.1)
```

Listing 24: Contoh Penggunaan Fungsi Training

Penggunaan fungsi testing:

```
acc, loss = test(X_test, y_test)

print(acc)
```

Listing 25: Contoh Penggunaan Fungsi Testing

13 Visualisasi Error dan Akurasi untuk 50 epoch

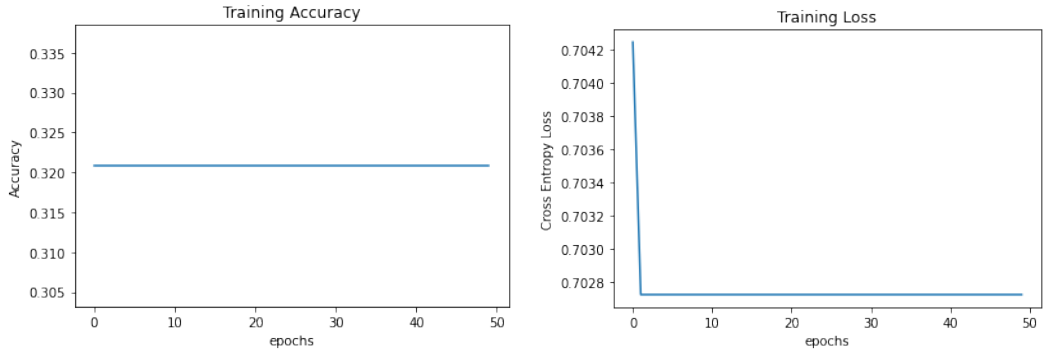

Figure 8: **Visualisasi akurasi dan error** h_{16}. Model MLP memiliki jumlah neuron pada hidden layer sebanyak 16 dengan learning rate 0.1.

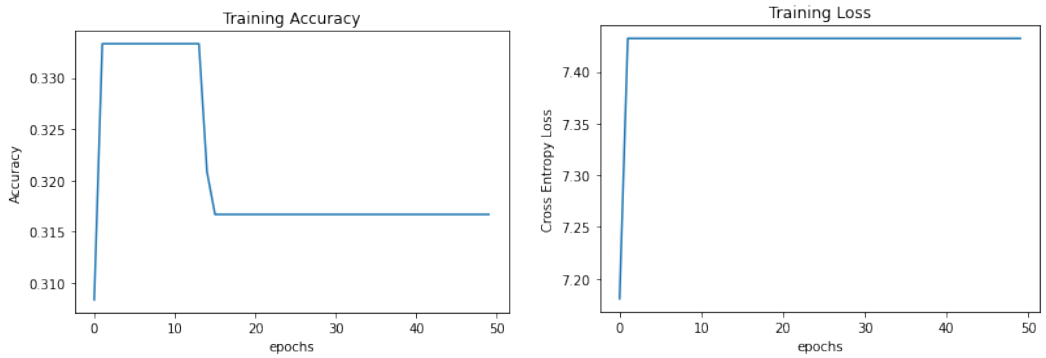

Figure 9: **Visualisasi akurasi dan error** h_{64}. Model MLP memiliki jumlah neuron pada hidden layer sebanyak 64 dengan learning rate 0.1.

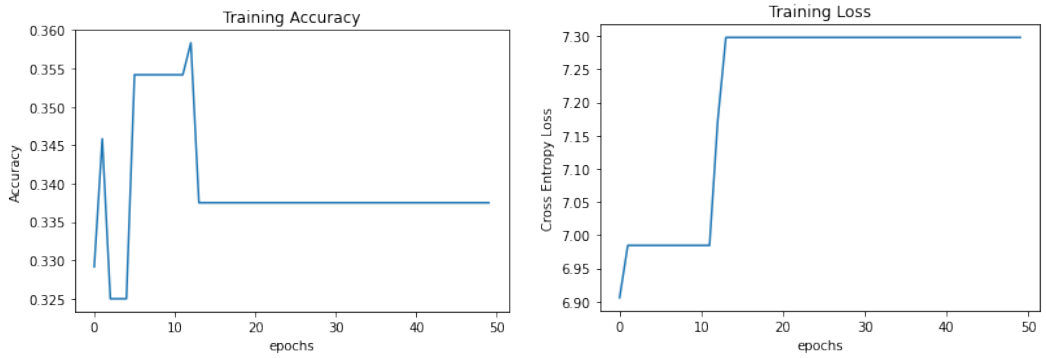

Figure 10: **Visualisasi akurasi dan error** h_{256}. Model MLP memiliki jumlah neuron pada hidden layer sebanyak 256 dengan learning rate 0.1.

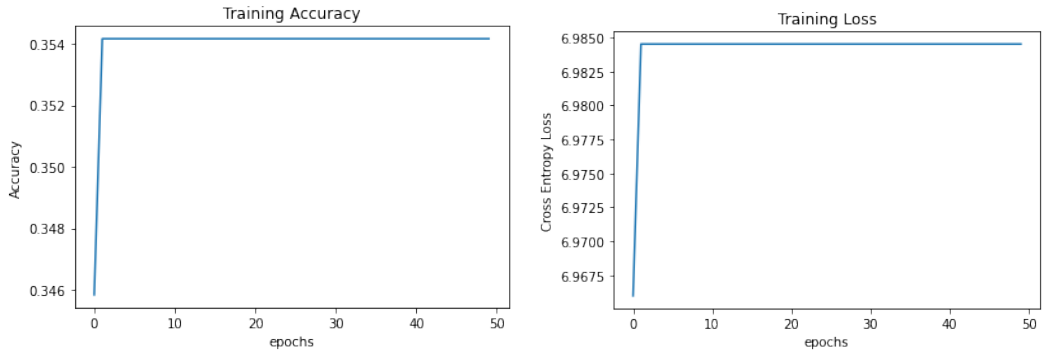

Figure 11: **Visualisasi akurasi dan error** h_{1024}. Model MLP memiliki jumlah neuron pada hidden layer sebanyak 1024 dengan learning rate 0.1.

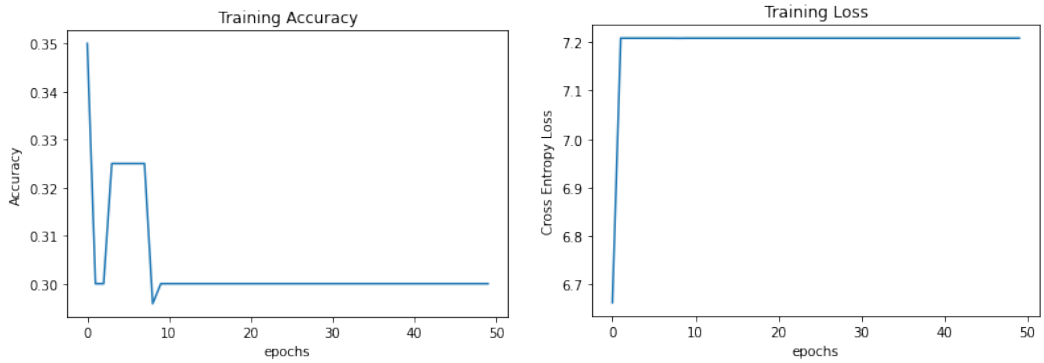

Figure 12: **Visualisasi akurasi dan error** h_{16}. Model MLP memiliki jumlah neuron pada hidden layer sebanyak 16 dengan learning rate 0.8.

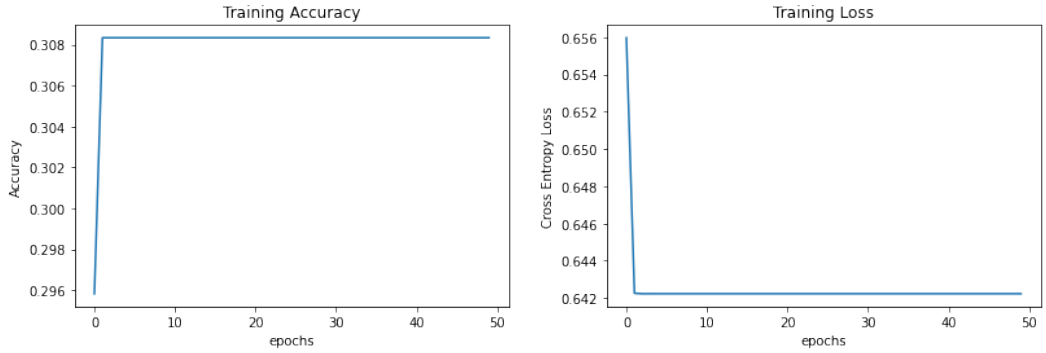

Figure 13: **Visualisasi akurasi dan error** h_{16}. Model MLP memiliki jumlah neuron pada hidden layer sebanyak 16 dengan learning rate 0.01.

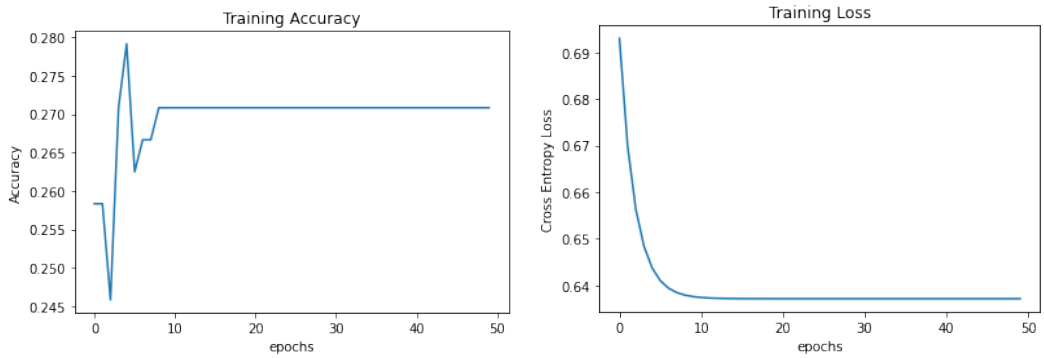

Figure 14: **Visualisasi akurasi dan error** h_{16}. Model MLP memiliki jumlah neuron pada hidden layer sebanyak 16 dengan learning rate 0.001.

https://github.com/Chrysophyt/MultilayerPerceptronFS/blob/main/Sigmoid.ipynb

14 Pembahasan

14.1 Model Multilayer Perceptron dengan Aktivasi Sigmoid

Model	Training Loss	Training Accuracy	Test Accuracy
MLP h_{16}	**0.702**	0.320	**0.33**
MLP h_{64}	7.432	0.316	**0.33**
MLP h_{256}	7.29	0.337	**0.33**
MLP h_{1024}	6.98	**0.354**	**0.33**

Table 1: **Hasil akurasi training dan testing dengan jumlah neuron pada hidden layer berbeda**. Semua model memakai learning rate konstan 0.1. Untuk akurasi pada dataset testing tidak ada perbedaan antar model MLP. Akurasi training pada model yang memiliki jumlah neuron pada hidden layer sebanyak 1024 atau h_{1024} memiliki akurasi terbesar yaitu 0.354. Waktu training model h_{16} hanya satu menit, dimana training model h_{1024} membutuhkan total 34 menit.

Model	Learning rate	Training Loss	Training Accuracy	Test Accuracy
MLP h_{16}	0.8	7.208	0.3	**0.33**
MLP h_{16}	0.1	0.702	**0.320**	**0.33**
MLP h_{16}	0.01	0.642	0.308	**0.33**
MLP h_{16}	0.001	**0.637**	0.270	**0.33**

Table 2: **Hasil Akurasi Training dan Testing dengan learning rate berbeda**. Sama seperti hasil sebelumnya untuk akurasi tidak ada perbedaan. Ketika learning rate 0.8 training loss menjadi lebih besar 7.208, sebaliknya ketika learning rate lebih kecil 0.001 training loss menjadi lebih kecil menjadi 0.637 tetapi akurasi turun drastis menjadi 0.270.

14.2 Model lain

Model	Training Loss	Training Accuracy	Test Accuracy
MLP sigmoid h_{16}	0.702	0.320	0.33
MLP softmax h_{512}	0.44	0.845	0.35
CNN	**0.001**	**1**	**0.516**

Table 3: **Hasil Akurasi Training dan Testing dengan model lain** (detail terdapat pada Other model.ipynb). Menggunakan Softmax dapat menaikan training pada MLP sampai 0.845, tetapi akurasi training hanya meningkat 0.35 sedangkan apabila menggunakan CNN, model akan overfit dengan akurasi training 1 dan akurasi tes mencapai 0.516. Apabila data ditambah lebih banyak, ini dapat memungkinkan untuk mencegah overfitting pada model CNN dan menghasilkan model yang lebih general yang memiliki akurasi lebih baik.

15 Tugas

Github repository terdapat pada : https://github.com/Chrysophyt/MultilayerPerceptronFS
Pembagian Tugas :

1. Chrystian: Mengurus GitHub, Dataset Preprocessing, Model dasar, Model lain, Visualisasi, Eksperimen, Penulisan Laporan.
2. Karunia Eka Putri: Feedforward, Fungsi Prediksi, Fungsi Akurasi, Eksperimen (hidden:16,64,256,1024 lr:0.8), Penulisan Laporan.
3. Widad Abida R: Fungsi Testing dan Training, Backward, Update Bobot, Eksperimen (hidden:128, lr:0.1, 0.8), Penulisan Laporan.