TUGAS BESAR BAGIAN B
IF3150 MACHINE LEARNING SEMESTER 2 2021/2022

Anggota:

Karlsen Adiyasa Bachtiar	13519001
Yudi Alfayat	13519051
Almeiza Arvin Muzaki	13519066
Roy H Simbolon	13519068

Tanggal Pengumpulan: 26 Maret 2022

TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2021

A. Implementasi Program

- Components
  - Activation

    Merupakan class yang berisi method-method fungsi aktivasi yang akan dipanggil pada method active sesuai dengan fungsi aktivasi yang sesuai dengan yang diperlukan. Method yang terdapat pada class aktivasi adalah method linear, sigmoid, relu dan softmax. Fungsi-fungsi aktivasi merupakan method static.

```
class Activation:
    LINEAR_ACTIVATION = "linear"
    SIGMOID_ACTIVATION = "sigmoid"
    RELU_ACTIVATION = "relu"
    SOFTMAX_ACTIVATION = "softmax"

    @staticmethod
    def linear(x) :
        return round(x)

    @staticmethod
    def sigmoid(x):
        value = 1 / (1 + math.exp(x*(-1)))
        return value

    @staticmethod
    def relu(x):
        return max(0.0, x)

    @staticmethod
    def softmax(x):
        return x

    @staticmethod
    def active(x, activation_function):
        if (activation_function == Activation.LINEAR_ACTIVATION):
            return Activation.linear(x)
        elif (activation_function == Activation.SIGMOID_ACTIVATION):
            return Activation.sigmoid(x)
        elif (activation_function == Activation.RELU_ACTIVATION):
            return Activation.relu(x)
        elif (activation_function == Activation.SOFTMAX_ACTIVATION):
            return Activation.softmax(x)
```

  - Layer

    Merupakan class yang menginisialisasi sebuah class layer.

- MiniBatchGradient

```
import math
import numpy as np
from components.Activation import Activation
from components.Layer import Layer


class MiniBatchGradient:
   def __init__(self, nodesArr, batchSize):
      self.batchSize = batchSize
      self.layers = [Layer(nbNodes, Activation.SIGMOID_ACTIVATION) for
nbNodes in nodesArr]
      self.weights = [np.random.rand(
         nodesArr[i]+1, nodesArr[i+1])-0.5 for i in range(len(nodesArr)-1)]

   def train(self, trainingData, trainingTarget, epochs, minCumulativeError,
learningRate):
      trainResult = {
         "epochs": epochs,
         "minErr": minCumulativeError,
         "learnRate": learningRate,
         "err": [],
         "acc": []
      }

      nbData = len(trainingData)

      for i in range(epochs):
         accuracy = 0
         cumulativeError = 0
         count = 0
         globalDeltaW = [np.zeros(
            (self.layers[i].nbNodes+1, self.layers[i+1].nbNodes)) for i in
range(len(self.layers)-1)]

         for j in range(nbData):
            inputs = trainingData[j]
            target = trainingTarget[j]

            output = self.feedForward(inputs)
            totalError = self.calcTotalError(output, target)
            cumulativeError += totalError

            localDeltaW = self.backwardProp(target, learningRate)

            for k in range(len(localDeltaW)):
```

```
                    globalDeltaW[k] += localDeltaW[k]

                if np.argmax(output) == np.argmax(target):
                    accuracy += 1

                if count == self.batchSize or i == epochs-1:
                    count = 0
                    for k in range(len(self.weights)):
                        self.weights[k] += globalDeltaW[k]
                    globalDeltaW = [np.zeros(
                        (self.layers[i].nbNodes+1, self.layers[i+1].nbNodes)) for i in
range(len(self.layers)-1)]

                count += 1

            trainResult["err"].append(cumulativeError)
            trainResult["acc"].append(accuracy/nbData)
            # print(f"Epoch{i+1} done! (err={cumulativeError},
acc={round(accuracy/nbData, 2)})", end="; ")
            if (i % 100 == 0 or i == epochs-1):
                print(f"e{i+1}(err={cumulativeError}, acc={round(accuracy/nbData,
2)})")

            # Threshold
            if cumulativeError <= minCumulativeError:
                break

        return trainResult

    def predict(self, instances):
        results = []
        for instance in instances:
            res = self.feedForward(instance)
            idx = np.argmax(res)
            results.append(idx)

        return results

    def calcTotalError(self, outputs, targets):
        errors = 0
        for i in range(len(targets)):
            errors += math.pow((targets[i]-outputs[i]), 2)

        return errors*0.5

    def feedForward(self, inputs, pr=False):
        if len(inputs) != self.layers[0].nbNodes:
            raise ValueError("Input error!!!")
        else:
            self.layers[0].outputs = np.array(inputs)
```

```python
        instance = inputs
        for i, layer in enumerate(self.layers[1:]):
            instance = np.append(instance, 1)  # biasnya
            val = np.dot(instance, self.weights[i])
            if pr:
                print(val, i)
            instance = layer.compute(val)

        return instance

    def backwardProp(self, targets, learningRate):
        deltaWeights = [np.zeros((self.layers[i].nbNodes+1,
self.layers[i+1].nbNodes))
                    for i in range(len(self.layers)-1)]

        for i in range(len(self.layers)-1, 0, -1):
            currLayer = self.layers[i]

            if i == len(self.layers)-1:
                for j in range(currLayer.nbNodes):
                    outputK = currLayer.outputs[j]
                    self.layers[i].deltas[j] = outputK * \
                        (1-outputK)*(targets[j]-outputK)

            else:
                for j in range(currLayer.nbNodes):
                    outputH = currLayer.outputs[j]
                    self.layers[i].deltas[j] = outputH * \
                        (1-outputH) * \
                        np.dot(self.weights[i][j], self.layers[i+1].deltas)

        for n in range(len(deltaWeights)):
            outputs = self.layers[n].outputs
            deltas = self.layers[n+1].deltas

            partialRes = []
            for output in outputs:
                for delta in deltas:
                    partialRes.append(output*delta)

            for delta in deltas:
                partialRes.append(delta)

            i = 0
            for row in range(len(deltaWeights[n])):
                for col in range(len(deltaWeights[n][row])):
                    deltaWeights[n][row][col] = partialRes[i]*learningRate
                    i += 1

        return deltaWeights
```

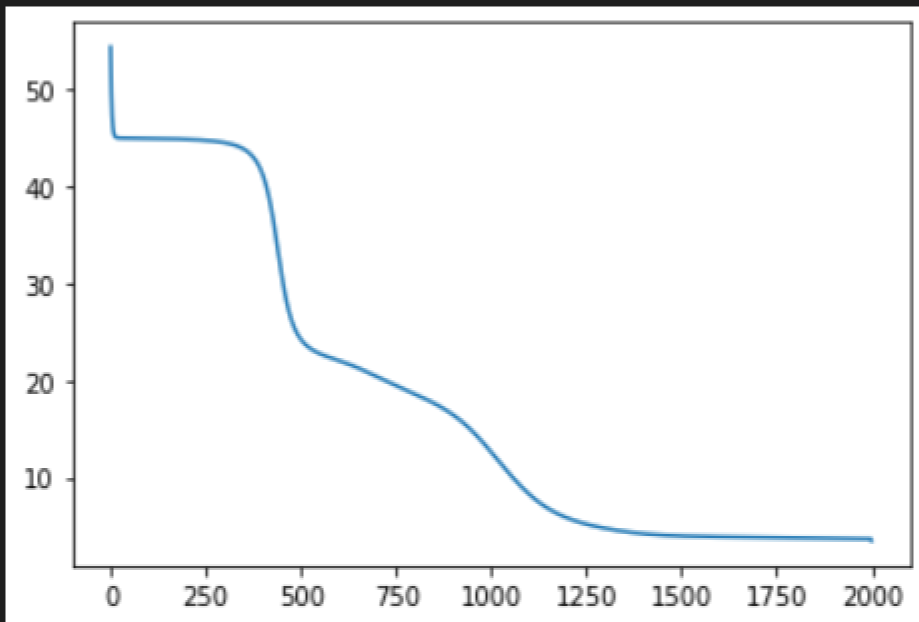
- Main
  Merupakan driver yang akan memanggil method-method components untuk melakukan implementasi pembelajaran mesin backpropagation dengan mini-batch gradient descent. Setelah load dataset Iris, miniBatchGradient membuat beberapa layer dan jumlah neuronnya dilanjutkan dengan trainModel. Di akhir main, dilakukan visualisasi untuk menampilkan model berupa grafik error dan akurasi hasil pembelajaran.

B. Hasil Pengujian

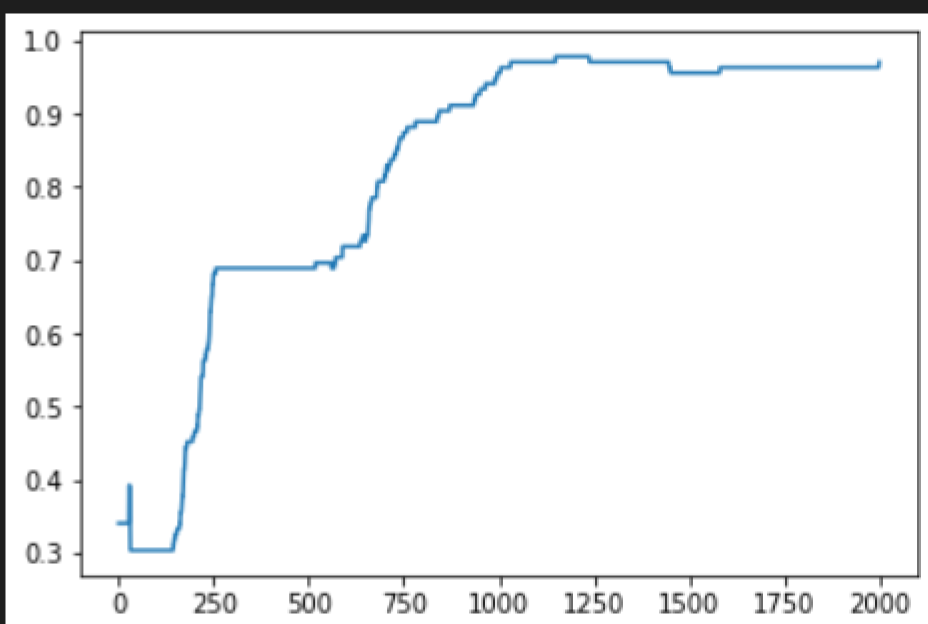
```
Begin training...
e0(err=54.419961813326736, acc=0.34)
e100(err=44.97675376505191, acc=0.3)
e200(err=44.884291823622895, acc=0.47)
e300(err=44.54022557278175, acc=0.69)
e400(err=41.200664753950896, acc=0.69)
e500(err=24.22012102333151, acc=0.69)
e600(err=22.028608474551213, acc=0.72)
e700(err=20.37333701126126, acc=0.81)
e800(err=18.576907280441237, acc=0.89)
e900(err=16.438557794245373, acc=0.91)
e1000(err=12.596002789556076, acc=0.96)
e1100(err=8.245196072654794, acc=0.97)
e1200(err=5.799849652481316, acc=0.98)
e1300(err=4.727120869802037, acc=0.97)
e1400(err=4.180754905999671, acc=0.97)
e1500(err=3.9464816712225734, acc=0.96)
e1600(err=3.858612437398909, acc=0.96)
e1700(err=3.8003891900496294, acc=0.96)
e1800(err=3.7430336835751734, acc=0.96)
e1900(err=3.6870419505452547, acc=0.96)
e1999(err=3.4498347180489253, acc=0.97)

Training done!!
Accuracy :  1.0
[2, 1, 0, 2, 0, 2, 0, 1, 1, 1, 2, 1, 1, 1, 1]
```

Visualisasi Error



Visualisasi Akurasi



C. Perbandingan dengan Hasil MLP Sklearn

```
Sklearn Result
****************************************************
Num of Data             : 150
Batch size              : 32
Epochs                  : 2000
Accuracy                : 0.9466666666666667
```

Berdasarkan hasil yang didapat kelompok kami, didapatkan hasil akurasi dengan menggunakan Backward Propagation dengan Mini Batch Gradient Descent yaitu 100%. Sedangkan dengan menggunakan MLP Sklearn menghasilkan akurasi 94.67%.

D.Pembagian Tugas

NIM	Pekerjaan
13519001	Laporan, membuat class MiniBatchGradient
13519051	Laporan, membuat main + testing + visualisasi
13519066	Laporan, membuat main + perbandingan MLP dengan Sklearn + visualisasi
13519068	Laporan, membuat class Activation dan Layer