IF4074 Pembelajaran Mesin

Mini-batch Gradient Descent (Feed Forward Neural Network)

Oleh:

Annisa Sekar Ayuningtyas	13516044
Muhammad Alif Arifin	13516078
Deborah Aprilia Josephine	13516152

PROGRAM STUDI SARJANA INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2019

DAFTAR ISI

DAFTAR ISI **2**

PENJELASAN KODE PROGRAM **3**

1.1. Kelas 3

1.1.1 Atribut 3

1.1.2 Fungsi 3

1.1.3 Algoritma 4

HASIL DAN ANALISIS EKSPERIMEN KLASIFIKASI DATASET **10**

2.1. Hasil Eksperimen 10

2.2. Analisis Hasil Eksperimen 12

PEMBAGIAN TUGAS KELOMPOK **14**

DAFTAR PUSTAKA **15**

PENJELASAN KODE PROGRAM

1.1. Kelas

1.1.1 Atribut

Kelas MiniBatch memiliki atribut sebagai berikut:

- __nb_nodes : jumlah *node* pada setiap *hidden layer*
- __hidden_layer : jumlah *hidden layer* pada setiap *network*
- __batch_size : jumlah data *latih* pada setiap *batch*
- __learning_rate : nilai *learning rate*
- __momentum : nilai momentum
- __epoch : jumlah *epoch*
- __errors : berisi nilai delta (*local gradient*) pada setiap *node*-nya
- __outputs : list yang berisi nilai predict setiap node pada hidden layer sampai output layer
- __weights : list yang berisi nilai *weight* untuk setiap pasangan *input-output node*
- __weights_bef : list yang berisi delta *weight* iterasi sebelumnya
- __batch_X : list yang berisi masukan data latih untuk *batch* yang dilatih
- __batch_y : list yang berisi label data latih untuk *batch* yang dilatih

1.1.2 Fungsi

Berikut adalah fungsi dalam kelas MiniBatch :

- __random_weights(self) : membangkitkan nilai *weights* dalam satu *batch*
- __sigmoid(self, v) : menjalankan fungsi sigmoid pada nilai v
- __psi_apostrophe(self, value) : value * (1 - value)
- __generate_batch(self) : membangkitkan network yang akan di train dalam satu batch
- __forward_pass(self) : menjalankan tahapan *forward pass* pada ANN, dengan *Feed Forward*
- __backward_pass(self) : menjalankan tahapan *backward pass* pada ANN, dengan *Backpropagation*
- __update_weights(self) : menjalankan tahapan *update weights* pada ANN

- fit(self, X, y) : melakukan *training* pada data *input* X dan data label y
- predict(self, X) : melakukan prediksi pada data *testing* X

1.1.3 Algoritma

Algoritma FFNN Mini-Batch Gradient Descent terbagi menjadi 2 tahap utama, yaitu inisialisasi dan *epoch*. Untuk setiap *epoch* akan melakukan *generate batch*, *forward pass*, *backward pass/backpropagation* dan *update weight*. Secara kasar, *pseudocode* dari FFNN MiniBatch Gradient Descent adalah sebagai berikut.

```
X = features
y = targets
batch_size = num of batch
epoch = num of epoch

gradient descent:
    // initialize
    init_weight()
    for i := 0 to epoch do
        batches = generate_batch(batch_size)
        for every batch in batches do
            forward_pass()
            backward_pass()
            update_weight()
```

Inisialisasi merupakan tahap yang bertanggung jawab untuk inisiasi *weight*. *Weight* diinisiasi secara acak dengan nilai -1 hingga 1. Setelah itu untuk setiap *epoch*, akan dilakukan pembangkitan *batch* sesuai dengan ukuran *batch*-nya, apabila jumlah *row* mod *batch_size* != 0 maka *batch* terakhir ukurannya akan kurang dari *batch_size*. Untuk setiap *batch* dilakukan *forward pass, backward pass* dan *update weight*.

a. *Forward Pass*

```
    def __forward_pass(self, is_train) :
        batch = self.__batch_X.values

        self.__outputs = [] # initialize output to zero

        for row_idx in range(len(self.__batch_X)) : # iterate for each row
            row = [] # for output in each row

            for layer_idx in range (self.__hidden_layer + 1) : # iterate for
```

```
each layer
                layer = [] # for output in each layer

                for node_idx in range (self.__n_nodes[layer_idx + 1]) : #
iterate for each node in output layer
                    node_v = 0

                    for input_idx in range(self.__n_nodes[layer_idx] + 1) : #
iterate for input node from input layer, +1 for
                        input_value = 0
                        if input_idx == 0 : # bias
                            input_value = 1
                        else :
                            if layer_idx == 0 : # first layer (input from
dataset)
                                input_value = batch[row_idx][input_idx - 1]
                            else : # input from output before
                                input_value = row[layer_idx - 1][input_idx -
1]
                                # if not is_train :

                        node_v += input_value *
self.__weights[layer_idx][input_idx][node_idx]

                    if layer_idx == self.__hidden_layer :
                        if is_train :
                            node_v = round(self.__sigmoid(node_v))
                            # node_v = self.__sigmoid(node_v)
                        else :
                            node_v = round(self.__sigmoid(node_v))
                    else :
                        node_v = self.__sigmoid(node_v)

                    layer.append(node_v)
                row.append(layer)
            self.__outputs.append(row)
```

Forward pass dilakukan untuk setiap *network* dalam satu *batch* dan menghasilkan *list* self.__outputs yang berisi nilai prediksi pada setiap node dari *hidden layer* hingga *output layer*. Dalam satu node output dilakukan pencarian nilai v dengan rumus :

$$v_j = b_j + x_1 . w_{1j} + x_2 . w_{2j} + .. + x_i . w_{ij}$$

j = node pada *layer output*/layer yang dituju

i = node pada *layer input*/ layer sebelumnya

x = nilai *node* pada *layer input*

w = nilai bobot pada pasangan *node layer input - output*

b = bias pada *node layer output/layer* yang dituju

Kemudian nilai v dikenakan fungsi aktivasi, dalam kasus ini fungsi aktivasinya adalah *sigmoid*, sehingga menghasilkan rumus :

$$y_j = \varphi(v_j) = \frac{1}{1 + exp(-v_j)}$$

y = nilai prediksi pada setiap *node* pada layer yang dituju

Self.__outputs berupa *multidimensional list* dengan dimensi pertama berupa *row/network*, kemudian dimensi yang kedua adalah *layer,* dan yang ketiga adalah *node.*

b. *Backward Pass*

```
    def __backward_pass(self) :
        # initialize errors to all zero
        self.__errors = []

        for idx, output in enumerate(self.__outputs) :
            temp_error = []
            # output layer
            o_idx = self.__hidden_layer
            o_predict = output[o_idx][0]
            temp_error.insert(0, [self.__psi_apostrophe(o_predict) *
(self.__batch_y[idx] - o_predict)]) # for error output layer

            # hidden layer
            # -2 dari -1 karena index dimulai dari 0
            #          -1 karena tidak pakai output dari layer output
            for i in range (len(output) -2, -1, -1) :

                # perkalian matriks
                matrix_error = np.matrix(temp_error[0])
                matrix_weight = np.matrix(self.__weights[i + 1])
```

```
                result = matrix_weight.dot(matrix_error.T)
                result = np.squeeze(np.asarray(result.T)).tolist()

                del result[0]

                temp_error.insert(0, list(map(lambda x, y:
self.__psi_apostrophe(x) * y, output[i], result)))

            # append output
            self.__errors.append(temp_error)
```

Backward pass dilakukan untuk menghitung *local gradient* dari nilai prediksi yang sudah dilakukan pada *forward pass*. Seperti namanya, *backward pass* dilakukan mundur mulai dari *output layer* hingga layer pertama dengan rumus sebagai berikut :

- Untuk *output layer*

$$\delta_i = o_i (1 - o_i) (t_i - o_i)$$

- Untuk *hidden layer*

$$\delta_i = o_i (1 - o_i) \sum_{j \varepsilon outputs} w_{ij} \delta_j$$

δ = nilai *local gradient* yang dihasilkan

o = nilai pada *node* yang dituju

i = *node* yang ingin dicari nilai *error*-nya

j = *node* pada *layer output/layer* setelah *node i*

__errors akan berisi seluruh delta (*local gradient*) yang didapatkan pada 1 *network* untuk tiap data pada *batch*.

c. *Update Weights*

```
def __update_weights(self) :
        temp_weights = self.__weights

        # delta weight
        delta_weights = []
        for i in range(len(self.__weights)) : # iterate layer
```

```
            delta_weight = []
            # iterate node input
            for j in range(len(self.__weights[i])):
                deltas = []
                # iterate node output
                for k in range(len(self.__weights[i][j])) :
                    delta = 0 # variable to sum all delta weight
                    # iterate row in a batch
                    for idx in range (len(self.__outputs)) :
                        self.__outputs[idx].insert(0,
                             self.__batch_X.iloc[idx].tolist())
                        self.__outputs[idx][i].insert(0, 1)
                        delta += (self.__momentum *
self.__weights_bef[i][j][k]) +
                                  (self.__learning_rate *
self.__errors[idx][i][k] *
                                   self.__outputs[idx][i][j])
                        del self.__outputs[idx][i][0]
                        del self.__outputs[idx][0]
                    deltas.append(delta)
                delta_weight.append(deltas)
            delta_weights.append(delta_weight)

        # update weight
        for i in range(len(self.__weights)) :
            for j in range(len(self.__weights[i])) :
                for k in range(len(self.__weights[i][j])) :
                    temp_weights[i][j][k] = self.__weights[i][j][k] +
                                            delta_weights[i][j][k]
        self.__weights_bef = self.__weights
        self.__weights = temp_weights
```

Update weights dilakukan setelah perhitungan *error* dari setiap *row data* dilakukan untuk setiap *batch* dan setiap *epoch.* Sebelum melakukan *update* nilai *weight,* pertama-tama hitung terlebih dahulu akumulasi *delta weight* dari setiap *row*. Perhitungan *delta weight* mengikuti formula berikut. Keterangan i adalah layer masukan, sedangkan j adalah layer keluaran. Setelah perhitungan *delta weight*, dilakukan *update weight* dengan menggunakan formula berikut.

$$\Delta w_{n(ij)} = (momentum * w_{n-1(ij)}) + (learning\_rate * \delta_{(j)} * o_{(i)})$$

$$w_{n(ij)} = w_{n-1(ij)} + \Delta w_{n(ij)}$$

Selain *update weight*, juga dilakukan *update* bias pada setiap *node*. Berikut formula yang digunakan untuk melakukan *update bias*.

$$\Delta b_{n(ij)} = (momentum * b_{n-1(ij)}) + (learning\_rate * \delta_{(j)} * 1)$$

$$b_{n(ij)} = b_{n-1(ij)} + \Delta b_{n(ij)}$$

HASIL DAN ANALISIS EKSPERIMEN KLASIFIKASI DATASET

2.1. Hasil Eksperimen

Eksperimen dilakukan pada dataset weather yang terdapat di pranala (https://storm.cis.fordham.edu/~gweiss/data-mining/weka-data/weather.arff) dengan total 13 *rows*. Dataset dituliskan dalam format CSV. Dataset terdiri dari kolom outlook, temperature, humidity, windy dan play. *Task* yang ditujukan pada dataset ini adalah mengklasifikasikan data ke dalam kelas Yes atau No pada fitur play.

Pada eksperimen ini, dilakukan beberapa *preprocessing* data. Fitur outlook dan windy dilakukan *one-hot encoding* untuk data sunny, overcast dan rainy. Fitur temperature dan humidity dilakukan proses normalisasi terhadap nilai minimum dan maksimum pada kolom. Berikut formula normalisasinya.

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Setelah dataset di-*preprocessing*, dataset kemudian di-*split* menjadi data *training* dan data *testing*. Kemudian data *training* di-*train* terhadap model MiniBatch, dan data *testing* diuji menggunakan model MiniBatch yang telah di-*train*. Hasil *testing* kemudian dievaluasi menggunakan metrik akurasi. Eksperimen dilakukan untuk mengukur hasil pengerjaan model. Pada eksperimen digunakan beberapa variabel terkontrol yaitu jumlah epoch, *learning rate* dan momentum. Berikut beberapa hasil eksperimen.

a. Jumlah epoch

   Parameter : nb_nodes = 5, hidden_layer = 4, batch_size = 3, learning_rate = 0,1, momentum = 0,001

Jumlah Epoch	Akurasi
10	57,14%
25	42,85%
50	28,57%

100	71,42%
200	71,42%
500	71,42%

b. Learning rate

Parameter : nb_nodes = 5, hidden_layer = 4, batch_size = 3, epoch = 25, momentum = 0,001

Learning Rate	Akurasi
0,1	42,85%
0,2	42,85%
0,3	42,85%
0,4	42,85%
0,5	57,14%
0,6	57,14%
0,7	57,14%
0,8	42,85%

c. Momentum

Parameter : nb_nodes = 5, hidden_layer = 4, batch_size = 3, epoch = 25, learning_rate = 0,5

Momentum	Akurasi
0,0005	42,85%
0,001	57,14%
0,005	57,14%
0,01	42,85%
0,05	42,85%

0,1	42,85%

d. Jumlah batch

Parameter : nb_nodes = 3, hidden_layer = 2, epoch = 25, learning_rate = 0,25, momentum = 0,005

Jumlah Batch	Akurasi
1	42,85%
2	71,42%
3	42,85%
4	57,14%
5	42,85%

2.2. Analisis Hasil Eksperimen

Dalam implementasinya, dengan berbagai variabel yang digunakan tidak berjalan linier dengan akurasi. Ada di mana akurasi meningkat, namun setelah di titik tertentu akurasi akan menurun. Selain itu jumlah dataset termasuk sedikit sehingga nilai dan akurasi yang dihasilkan fluktuatif. Secara keseluruhan hasil dari model FFNN Mini Batch Gradient Descent memiliki akurasi yang rendah. Hal tersebut dapat dikarenakan beberapa hal, seperti data yang kurang, data yang tidak ternormalisasi dengan baik, urutan data yang tidak pas, fungsi aktivasi yang kurang sesuai dan inisiasi awal *weight* yang kurang beruntung.

Untuk eksperimen *epoch* semakin banyak *epoch* semakin bagus karena *learning rate* yang cukup kecil, yaitu 0,1. Ketika *learning rate* kecil dibutuhkan jumlah *epoch* yang banyak agar dapat menuju konvergen. Ketika *epoch* terlalu kecil, perpindahan yang dilakukan tidaklah banyak sehingga kemungkinan masih tidak terlalu jauh dari posisi awal (seperti inisiasi *weight*).

Untuk eksperimen *learning rate*, akurasi tertinggi dicapai ketika *learning rate* sebesar 0,5 hingga 0,7. Eksperimen ini kurang representatif untuk ditarik kesimpulan karena jumlah epoch yang

terlalu kecil. Hal tersebut dimungkinkan ketika *learning rate* < 0,5 (terlalu rendah), MBGD tidak berlatih dengan baik karena pergerakan terlalu kecil (perubahan tidak signifikan) sehingga membutuhkan *epoch* yang lebih banyak agar hasil menjadi konvergen. Ketika *learning rate* > 0,7 (terlalu besar), MBGD akan sulit untuk mencapai konvergen karena pergerakan terlalu besar (perubahan terlalu besar). Seharusnya ketika jumlah epoch tepat, maka ketika *learning rate* terlalu tinggi tidak akan konvergen sehingga tidak menghasilkan nilai yang baik.

Untuk eksperimen momentum, sama seperti eksperimen *learning rate* tidak memberikan hasil yang dapat ditarik kesimpulan yang baik karena jumlah epoch yang terlalu kecil. Namun berdasarkan hasil, dapat dilihat bahwa momentum cukup berpengaruh untuk meningkatkan akurasi pada model karena momentum menggunakan *weight* di batch sebelumnya.

Untuk eksperimen *batch size*, ketika *batch* terlalu besar yang lebih besar ada penurunan yang signifikan dalam kualitas model, yang diukur dengan kemampuannya untuk menggeneralisasi.

Dari semua eksperimen yang ada, terdapat kemungkinan hasil eksperimen kurang maksimal dikarenakan data yang kurang banyak untuk melakukan *deep learning*.

PEMBAGIAN TUGAS KELOMPOK

Annisa Sekar Ayuningtyas (13516044)	Muhammad Alif Arifin (13516078)	Deborah Aprilia Josephine (13516152)
Implementasi fungsi forward_pass	Implementasi struktur kelas	Implementasi fungsi update_weights, predict
Implementasi fungsi sigmoid	Implementasi fungsi backward_pass, psi_apostrophe, random_weights, fit, generate_batch	Implementasi Notebook untuk melakukan eksperimen
Debugging	Debugging	Debugging
Melakukan eksperimen	Melakukan eksperimen	Melakukan eksperimen
Membuat laporan	Membuat laporan	Membuat laporan

DAFTAR PUSTAKA

Dosen Teknik Informatika ITB. *IF3170 Intelegensi Buatan: Artificial Neural Network.*

Dosen Teknik Informatika ITB. *IF4074 Pembelajaran Mesin Lanjut: Overview Regresi dan Artificial Neural Network*.

Brownlee, Jason. 2017. *A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size*. Diaskes dari https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/ pada tanggal 28 September 2019.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. https://arxiv.org/abs/1609.04836 :