Greedy Implementation of Neural Networks Gradient Descent

Abram Perdanaputra - 13516083[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]abram.perdanaputra@gmail.com

***Abstract*—. Artificial General Intelligence or more commonly Artificial Intelligence (AI) is gaining its fame because how big is its potential to solve the currently unsolved problems. AI can be extended into many kind of field such as medical, transportation, education, civil, and many other. AI is using Machine Learning (ML) algorithms to learn and predict solution. The goal of common ML models is how to minimize the error prediction. Without us knowing, one of the AI model which is Neural Network is using Greedy technique to minimize its loss function. Every training sessions, the Neural Network wil forward pass and do backpropagation using Gradient Descent —which is a Greedy technique — to adjust the weights to minimize the loss function.**

***Keywords*—Greedy, Neural Network, Gradient Descent, Artificial Intelligence.**

I. Introduction

These days, Artificial Intelligence (AI) is gaining popularity. Many companies boasts that their product is AI-powered so that it could be superpower to common people. Products such as self driving cars, personal assistant, videos / article recommenders, image recognition, pattern generator, and many other kind of products that is using AI to work better and help humans better.

The society is having many different perceptions of AI. Some people doesn't even know how AI works, others are very excited on how it is helping human lives, others work to use AI to save many human lives, but under the hood, all of AI is pure and simple calculus math. One common and very basic AI model is the Artificial Neural Network. It's the base model that inspires other models such as Convolutional Neural Network, Recurrent Neural Network, Associative Neural Network, and many other types of Neural Network (NN). How's Neural Network able to solve problems?

II. Theory

When trying to solve problem, humans tend to see the best solution available to pick, but in sequential problem, this kind of selection method doesn't always give the optimum solution we're looking for. That's how Greedy technique sees things when trying to solve a certain problem. Greedy approach suggests constructing solution through a sequence of steps, each each expanding a partially constructed solution obtained so far, until the complete solution to the problem is reached [1]. On each step, the solution must be feasible — satisfying the problem constraints—, locally optimal — best local choice that can be taken—, and irrevocable — once its taken, there's no turning back. These requirements explains Greedy's name. On each step, we have to be "greedy" to grab the best alternative possible solution available hoping that the sequence of locally optimal solution gives us the global solution to the entire problem.

By this time, there are many Greedy technique algorithms that able to find the optimal solutions at all times such as Prim and Kruskal's algorithm to find the minimum spanning tree given a graph, Dijkstra's algorithm to find the shortest path to all other vertices given a graph and it's source vertex, and Huffman Trees and Codes to compress a sequence of strings. But there are many problems that Greedy technique couldn't solve such as Coin Partition Problem, Traveling Salesperson Problem (TSP), Integer Knapsack Problem, and many other. In this Neural Network problem, Greedy technique cannot be guaranteed to give the optimal solution at all times.

Before diving further, we have to describe what NN is and define the problem we're trying to solve. Neural Network is a sequence of layers that consists of nodes, weights to connect one node to another, and optionally bias for each node. Here, we define a node is as simple as something that holds a number. The structure of a simple NN is given in Figure 1.1.

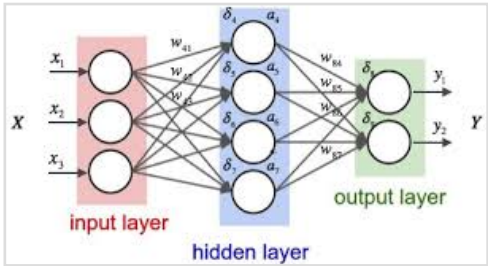

Figure 2.1 Neural Network structure.
Source : https://medium.com/@curiousily/tensorflow-for-hackers-part-iv-neural-network-from-scratch-1a4f504dfa8

Figure 1.1 shows that a NN consists of one input layer, one output layer, and hidden layer with any arbitrary number of layers inside it. It also shows that every node have some kind of connection to every node in the next layer called weights. It represents how strong is a node effecting a node in the next layer. For example, w_{41} represents how's the node 1 effects the node 4. The weight could be negative—the node 4 doesn't like if the node 1 is high—, positive— the node 4 is higher if the node 1 is high—, or zero—node 1 doesn't affect node 4 at all.

A NN receives input and will place the inputs to the input layer to become the nodes value. Here we call the node's value as activation. To calculate the activation of a node, simply just calculate the "weighted sum" of its input, add the bias, and decide whether to activate of not activate the node.

(a) $\delta_4 = w_{41} * x_1 + b_{41}$

(b) $a_4 = \sigma(\delta_4)$

Figure 2.2 (a) Weighted sum of a node (b) Activation of a node

In NN, we decide to activate or not activate a node by applying a function called *activation function*. Activation function could be any function you would like, but the most common ones are Sigmoid function, ReLU function, Softmax function, and Step function. That's all the basics needed for knowing how is Neural Network using a Greedy technique to solve problems.

III. Greedy Implementation on Neural Network

Every Machine Learning model needs a set of data to learn from. In Machine Learning, there's two kind of learning process, Supervised Learning and Unsupervised Learning. The difference between those two are Supervised Learning tells the model what's the correct result from a given input. Unsupervised Learning doesn't tell what's the expected output. The problem domain of Supervised and Unsupervised learning is given to the reader for research. In this problem, we're using Supervised Learning to train the Neural Network model to tackle a given problem.

One other essential component of a Neural Network model is the loss function. A loss function is how we measure the performance of how's the model is doing, is it nearly correct or is it giving us utter mess. We can define many kind of loss functions, but the most common ones are Cross Entropy, Mean Squared Error, Hinge Loss, and Huber Loss. For simplicity sake, we're using Mean Squared Error loss function for easy use.

$$c = \frac{1}{n}\sum (y_i - a_i)^2$$

Figure 3.1 Mean Squared Error loss function.
Source : author

To compute the error, simply find the difference between every expected output and the output layer activation and square those values. Keep doing it for all the nodes in output layer and sum all the squared difference. Divide the sum by the number of nodes in the layer. This gives the mean squared error of the output layer given an expected output.

We already reviewed how the NN performance is evaluated. Now we can continue how Greedy technique is used on Neural Networks. First, we do forward pass to find the output layers activation. The activation of the output layer is the NN's prediction. When we have the NN's prediction for a given input, we measure the loss/error of the NN by using the loss function. The training sequence of a NN is evaluate the activation, count the loss function, and adjust the weights to minimize the loss function.

Here goes the Greedy technique. The problem is we need to find a global minimum of the loss function. To adjust the weights, we compute the the gradient of the loss function. We know from Figure 3.1 that our Mean Squared Error loss function has two random variables, so we need to find how's each parameter affect the loss function. For example, we compute the gradient of loss function over w_i to find in what w_i direction we have to go to minimize the loss function faster. We'll cover the details in the upcoming section below.

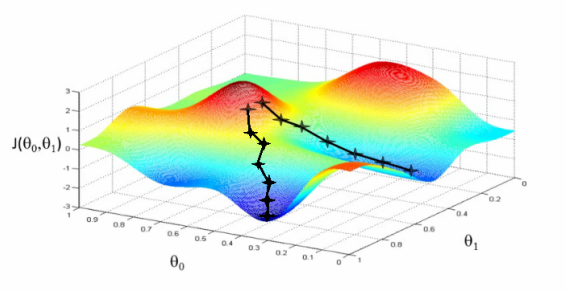

Figure 3.2 Gradient descent example.
Source : https://medium.com/ai-society/hello-gradient-descent-ef74434bdfa5

In the last section, we're going to move to the direction that minimizes our loss function. This tells us that we need to determine how much are we going to move in that direction. Now we introduce a *learning rate*. Learning rate is the magnitude of how many units should we move from a current point to the direction that makes the loss function minimum faster.

From Figure 3.2, we may realize that this kind of Greedy technique would give a different answer depending on the starting value of the loss function. We can see that a different starting value could move to a different minima point. One goes to a local minima, the other goes to the global minima. We wanted the loss function to be the smallest possible value, that means we wanted the loss function to go to the global minima all the time, but because of the Greedy technique implementation, this is not possible. There's no guarantee that this implementation is always minimizing the loss function. Next, we'll cover the details of the Greedy technique used in Gradient Descent to minimize the loss function.

IV. Gradient Descent

In the previous section, we've covered on how the Greedy technique is implemented in a Neural Network model. At every point it stops, we calculate the direction that makes the Loss function decreases the fastest without knowing is this going to the local minima or the global minima. How's exactly this process was done?

Gradient Descent is used while training a machine learning model. It is an optimization algorithm, based on a convex function, that tweaks it's parameters iteratively to minimize a given function to its local minimum. It is simply used to find the values of a functions parameters (coefficients) that minimize a cost function as far as possible. You start by defining the initial parameters values and from there on Gradient Descent iteratively adjusts the values, using calculus, so that they minimize the given cost-function. But to understand it's concept fully, you first need to know what a gradient is [2].

According to Lex Fridman from MIT, gradient is a measure of how much the output of a function changes if you change the input a little bit. It simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. Said it more

mathematically, a gradient is a partial derivative with respect to its inputs.

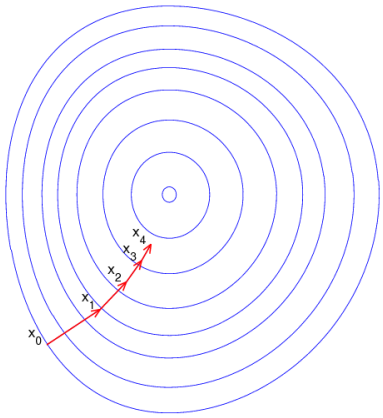

Figure 4.1 Gradient visualization.
Source : https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0

From Figure 4.1, we can imagine a blind man who wants to descend to a valley, with fewest steps possible. He just start walking the valley by taking big steps to the steepest direction, which he can do, as long as he's not near the bottom. As he come further to the bottom, he will do smaller and smaller steps, since he doesn't want to overshoot it. This process can described mathematically by using gradient. Imagine the red arrow is the step of the man with the goal to get to the minimum point of the valley.

$$b = a - \gamma \nabla f(a)$$

Figure 4.2 Gradient descent
Source : *author*

In Figure 4.2, b represents the next position of the man, while a represent the man's current position, and the *minus gamma times the gradient term* is the direction of the steepest descent. The formula above suggests what's the new position the man should go.

For example, Figure 3.2 represents a two dimensional function we're trying to minimize. First, we need to compute the directional derivative of the function by calculating this formula.

$$\nabla J(\theta_0, \theta_1) = \frac{\partial J}{\partial \theta_0}\mathbf{i} + \frac{\partial J}{\partial \theta_1}\mathbf{j}$$

Figure 4.3 Directional derivative of a function.
Source : *author*

This formula is computing how much should it move to the θ_0 direction and θ_1 direction and results a vector in which direction should it move to minimize the function. This formula could change depending on how many random variables the function we're trying to minimize has. For the Mean Squared Error function, we have two parameters to tweak, just like the one in Figure 3.2. Now, we've got to know what is gradient descent and how does it work. Next, we'll cover on how does this gradient descent is training the Neural Network through a process called Backpropagation.

V. BACKPROPAGATION

Backpropagation, short for "backward propagation of errors," is an algorithm for supervised learning of artificial neural networks using gradient descent. Given an artificial neural network and an error function, the method calculates the gradient of the error function with respect to the neural network's weights. It is a generalization of the delta rule for perceptrons to multilayer feedforward neural networks [3].

The "backwards" part of the name stems from the fact that calculation of the gradient proceeds backwards through the network, with the gradient of the final layer of weights being calculated first and the gradient of the first layer of weights being calculated last. Partial computations of the gradient from one layer are reused in the computation of the gradient for the previous layer. This backwards flow of the error information allows for efficient computation of the gradient at each layer versus the naive approach of calculating the gradient of each layer separately [3].

In this example case, we're going to use the Neural Network from Figure 2.1. This NN consists of three nodes input layer, one hidden layer with four nodes, and two nodes output layer. We define a superscript as the mark of which layer it belongs, such as a^i as the activation of the input layer a^h as the activation of the hidden layer a^o as the activation of the output layer. The same apply with the w for weights and δ for the weighted sum.

$$s_k^h = w_{k1}^h * a_1^i + w_{k2}^h * a_2^i$$
$$\mathbf{s^h} = \mathbf{w^h a^i}$$
$$a_k^h = \sigma(s_k^h)$$
$$\mathbf{z^h} = \sigma(\mathbf{s_k^h})$$
$$s_k^l = w_{k1}^l * a_1^h + w_{k2}^h * a_2^h + w_{k3}^h * a_3^h$$
$$\mathbf{s^l} = \mathbf{w^l a^h}$$
$$a_k^l = \sigma(s_k^l)$$
$$\mathbf{a^h} = \sigma(\mathbf{s_k})$$

Figure 5.1 Equations of each computation
Source : *author*

To do backpropagation, we have to adjust all the weights, meaning we have to adjust the weights of the output layer and the weights of the hidden layer by computing the partial derivative of the mean squared error function over the output layer weights and the hidden layer weights. We compute each partial derivative by calculating the equations in Figure 5.2.

$$\frac{\partial \mathbf{E}}{\partial \mathbf{w}^l} = \frac{\partial \mathbf{E}}{\partial \mathbf{a}^l}\frac{\partial \mathbf{a}^l}{\partial \mathbf{s}^l}\frac{\partial \mathbf{s}^l}{\partial \mathbf{w^l}}$$
$$\frac{\partial \mathbf{E}}{\partial \mathbf{w}^h} = \frac{\partial \mathbf{E}}{\partial \mathbf{a}^l}\frac{\partial \mathbf{a}^l}{\partial \mathbf{s}^l}\frac{\partial \mathbf{s}^l}{\partial \mathbf{a}^h}\frac{\partial \mathbf{a}^h}{\partial \mathbf{s}^h}\frac{\partial \mathbf{s}^h}{\partial \mathbf{w}^h}$$

Figure 5.2 Equations of partial derivatives.
Source : *author*

Next we compute the delta of the weights we need to adjust by subtracting the current weight with the product of the learning rate and each of the partial derivative.

$$\mathbf{w}^h = \mathbf{w}^h - \gamma \frac{\partial \mathbf{E}}{\partial \mathbf{w}^h}$$

$$\mathbf{w}^l = \mathbf{w}^l - \gamma \frac{\partial \mathbf{E}}{\partial \mathbf{w}^l}$$

Figure 5.3 Weights adjustment.
Source : author

VI. Implementation

After all we've discussed before, let's try to implement this method and try to build something intelligent. In this section, we're going to use Python and Jupyter Notebook. Let's try and build a Neural Network that accepts two integer inputs and outputs those two integers in sorted descending order.

In this example, we're going to implement the Neural Network using matrix and vector. We define layers as a one dimensional array or vector and the weights as two dimensional array or matrix. Each of the elements corresponds with one particular component in the Neural Network.

First of all, we need to import the libraries that we need. In this example, we're going to use NumPy for mathematical library to do all the matrix operation and Python random library to generate random data.

```
import numpy as np
import random as rand
```

Figure 6.1 Import statements.
Source : author

Next, define all the functions that we need for the activation functions. Here, we're using ReLU activation functions that will output *max(x, 0)* and it's derivative.

```
# Activation Functions
def relu(x):
    for i in range(0, len(x)):
        for k in range(0, len(x[i])):
            if x[i][k] < 0:
                x[i][k] = 0
    return x

def relu_derivative(x):
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = 1 if x[i, j] > 0 else 0
    return x
```

Figure 6.2 Activation functions.
Source : author

After defining all the needed functions, we're going to generate the data. Since this is a supervised learning, we need to generate the inputs and the correct output for each input. We're going to generate two random integer and make a tuple. For each tuple in the list, we're going to generate a tuple that's already in the sorted descending order. For this example we're generating 10000 sample data points.

```
# Generate training data
TRAIN_DATA_SIZE = 10000
inputs = [(rand.randrange(0, 50),
           rand.randrange(0, 50))
          for i in range(TRAIN_DATA_SIZE)]
labels = [(max(inputs[i][0], inputs[i][1]),
           min(inputs[i][0], inputs[i][1]))
          for i in range(TRAIN_DATA_SIZE)]

inputs = np.asarray(inputs)
labels = np.asarray(labels)
```

Figure 6.3 Generate training data.
Source : author

We also need to define the hyperparameters such as epoch (number of iterations), batch size (number of data that will be computed at a particular time), learning rate, decay factor for the learning rate to make the steps smaller each iteration so it won't overshoot, and number of neuron or nodes for each layer.

```
# Hyperparameters
EPOCH = 500
BATCH_SIZE = 100
DECAY_FACTOR = .99
NEURON_X = 2
NEURON_H = 5
NEURON_Y = 2

LEARNING_RATE = .00005
```

Figure 6.4 Hyperparameters.
Source : author

Now we're ready to initialize the network. We're going to initialize all the weights with a random uniform variable. This is the factor that makes the Neural Network doesn't always get to the global minima. All depends on the starting point of the error function.

```
# Weights initialization
weights_h = np.random.uniform(size=(NEURON_H, NEURON_X))
weights_y = np.random.uniform(size=(NEURON_Y, NEURON_H))
```

Figure 6.5 Weights initialization.
Source : author

Time to train the network. We're going to update all the weights and use the entire dataset 500 times. For each epoch, we're going to consume the dataset per batch with 100 datum each batch. This could make the learning more efficient since the computation is smaller than if we use the entire dataset.

If we execute the program from the beginning to this point, the network will start to learn from the given dataset. As we can see on Figure 6.7, the network's error is decreasing every epoch, this means that the network is actually learning and becoming more intelligent to solve this problem.

```
# Training

for i in range(EPOCH):
    LEARNING_RATE *= DECAY_FACTOR
    for j in range(len(inputs) // BATCH_SIZE):
        # Forward pass
        start_idx = int(j * BATCH_SIZE)
        curr_x = inputs[start_idx:(start_idx + BATCH_SIZE)].T
        curr_y = labels[start_idx:(start_idx + BATCH_SIZE)].T

        sum_h = np.dot(weights_h, curr_x)
        act_h = relu(sum_h)
        sum_y = np.dot(weights_y, act_h)
        act_y = relu(sum_y)

        # Compute error
        if i % 10 == 0 and j == 0:
            diff = act_y - curr_y
            loss = (diff * diff).sum() / BATCH_SIZE
            print("Epoch: {} Learning rate: {} Error: {}".format(
                i,
                LEARNING_RATE,
                loss))

        # Backprop for hidden layer
        d_act_y = act_y - curr_y
        del_y = relu_derivative(act_y) * d_act_y
        for row in range(NEURON_Y):
            weights_y[row] -= (act_h * del_y[row]).sum(axis=1)
                              * LEARNING_RATE / BATCH_SIZE

        d_act_h = np.dot(weights_y.T, del_y)
        del_h = relu_derivative(act_h) * d_act_h
        for row in range (NEURON_H):
            weights_h[row] -= (curr_x * del_h[row]).sum(axis=1)
                              * LEARNING_RATE / BATCH_SIZE
```

Figure 6.6 Training the network.
Source : author

```
Epoch: 0 Learning rate: 4.9500000000000004e-05 Error: 12029.765058511819
Epoch: 10 Learning rate: 4.4766912712935834e-05 Error: 103.12655604151288
Epoch: 20 Learning rate: 4.0486393411062934e-05 Error: 44.97557976782762
Epoch: 30 Learning rate: 3.661516848271988e-05 Error: 4.286397344849025
Epoch: 40 Learning rate: 3.3114102049199184e-05 Error: 0.23097586659887576
Epoch: 50 Learning rate: 2.9947800323308063e-05 Error: 0.022246357171132325
Epoch: 60 Learning rate: 2.7084253798342694e-05 Error: 0.007223284529186861
Epoch: 70 Learning rate: 2.449451365021026e-05 Error: 0.004234413441578648
Epoch: 80 Learning rate: 2.215239908130864e-05 Error: 0.0028306558177144158
Epoch: 90 Learning rate: 2.003423264757704e-05 Error: 0.0019902412470130704
Epoch: 100 Learning rate: 1.811860089302485e-05 Error: 0.0014498922600468464
Epoch: 110 Learning rate: 1.638613787189019e-05 Error: 0.0010889195362000907
Epoch: 120 Learning rate: 1.4819329369604e-05 Error: 0.0008404519639259052
Epoch: 130 Learning rate: 1.3402335845843706e-05 Error: 0.0006648873844349416
Epoch: 140 Learning rate: 1.2120832302229013e-05 Error: 0.0005378810168968311
Epoch: 150 Learning rate: 1.0961863468323617e-05 Error: 0.0004440259085774897
Epoch: 160 Learning rate: 9.913712829445722e-06 Error: 0.0003733194507582813
Epoch: 170 Learning rate: 8.965784179735527e-06 Error: 0.0003191148402089454
Epoch: 180 Learning rate: 8.108494500550326e-06 Error: 0.0002768995365777777
Epoch: 190 Learning rate: 7.333177081605183e-06 Error: 0.0002435486337927703
Epoch: 200 Learning rate: 6.631993905469105e-06 Error: 0.0002168580632277782 4
Epoch: 210 Learning rate: 5.997856409673895e-06 Error: 0.0001952465546468540 7
Epoch: 220 Learning rate: 5.424353825385735e-06 Error: 0.0001775615817522579
Epoch: 230 Learning rate: 4.905688368184296e-06 Error: 0.0001629506801933012 6
Epoch: 240 Learning rate: 4.436616625765069 5e-06 Error: 0.0001507746437848890 4
Epoch: 250 Learning rate: 4.012396550027973 6e-06 Error: 0.0001405480256237715
Epoch: 260 Learning rate: 3.628739517672466 4e-06 Error: 0.0001318977393186801 6
Epoch: 270 Learning rate: 3.281766974659073e-06 Error: 0.0001245338465729617
Epoch: 280 Learning rate: 2.9679712262375383e-06 Error: 0.0001182286703090995 9
Epoch: 290 Learning rate: 2.6841799761511304e-06 Error: 0.0001128016744214411 3
Epoch: 300 Learning rate: 2.427524256528643e-06 Error: 0.0001081083897833492 6
Epoch: 310 Learning rate: 2.1954094242535797e-06 Error: 0.0001040322142288485 1
Epoch: 320 Learning rate: 1.985488930600337e-06 Error: 0.0001004782774894977 7
Epoch: 330 Learning rate: 1.7956405989633447e-06 Error: 9.736880601457257e-05
Epoch: 340 Learning rate: 1.623945170860522e-06 Error: 9.463958847995236e-05
Epoch: 350 Learning rate: 1.4686669033233662e-06 Error: 9.223725692269147e-05
Epoch: 360 Learning rate: 1.328236021524341e-06 Error: 9.01171778513293e-05
Epoch: 370 Learning rate: 1.2012328492476225e-06 Error: 8.824180353147296e-05
Epoch: 380 Learning rate: 1.0863734567713030e-06 Error: 8.657937331987646e-05
Epoch: 390 Learning rate: 9.824966810693189e-07 Error: 8.510288337910305e-05
Epoch: 400 Learning rate: 8.885523871147342e-07 Error: 8.378926370390242e-05
Epoch: 410 Learning rate: 8.035908516128497e-07 Error: 8.261871643414656e-05
Epoch: 420 Learning rate: 7.267531618397213e-07 Error: 8.157418050370041e-05
Epoch: 430 Learning rate: 6.572625325238165e-07 Error: 8.064089589568736e-05
Epoch: 440 Learning rate: 5.94416452989431e-07 Error: 7.98060469209208e-05
```

Figure 6.7 Training result.
Source : author

Now we've finished training the network and we should test how does this network perform against input it hasn't seen. We've trained the network with dataset with range 0-50, now we're testing the network against a datum with range 0-500.

```
# Forward Pass
# Test Forward pass
i = rand.randrange(0, len(inputs))
nx = np.asarray((rand.randrange(0, 500), rand.randrange(0, 500))).T
ny = np.asarray((max(nx[0], nx[1]), min(nx[0], nx[1]))).T
act_h = reluld(np.dot(weights_h, nx))
act_y = reluld(np.dot(weights_y, act_h))


# Compute error
diff = ny - act_y
loss = (diff ** 2).sum()
print("x1 = [{} {}] \nr => [{} {}], \nGT = {}, \nloss = {}"
      .format(
    nx[0].sum(),
    nx[1].sum(),
    int(round(act_y[0].sum())),
    int(round(act_y[1].sum())),
    ny,
    loss))
```

Figure 6.8 Testing the network.
Source : author

If we run this snippet of code in Figure 6.8 with Jupyter Notebook, we're getting a result like in Figure 6.9.

```
x1 = [85 219]
r => [219 85],
GT = [219  85],
loss = 2.870706308851896e-07
----------------------------------------------------------
x1 = [180 130]
r => [180 130],
GT = [180 130],
loss = 9.10813736781239e-06
----------------------------------------------------------
x1 = [467 397]
r => [467 397],
GT = [467 397],
loss = 0.0004892260214117144
----------------------------------------------------------
x1 = [373 22]
r => [373 22],
GT = [373  22],
loss = 0.003179772850127567
----------------------------------------------------------
x1 = [284 466]
r => [466 284],
GT = [466 284],
loss = 1.1995170084589e-05
----------------------------------------------------------
x1 = [332 75]
r => [332 75],
GT = [332  75],
loss = 0.0013229813183569803
----------------------------------------------------------
x1 = [107 355]
r => [355 107],
GT = [355 107],
loss = 2.9338062140188466e-06
----------------------------------------------------------
x1 = [392 432]
r => [432 392],
GT = [432 392],
loss = 6.224017223705932e-05
----------------------------------------------------------
x1 = [213 312]
r => [312 213],
GT = [312 213],
loss = 9.84116102703184e-06
----------------------------------------------------------
```

Figure 6.6 Testing results.
Source : author

As we can see, the network performs well in even the data it hasn't seen. It gives the exact same output with the test label. But this is only nine test, we need to validate the network using even more range and even larger dataset.

```
# Generate testing data
TEST_DATA_SIZE = 10000000
inputs_test = [(rand.randrange(0, 10000),
                rand.randrange(0, 10000))
               for i in range(TEST_DATA_SIZE)]
labels_test = [(max(inputs_test[i][0], inputs_test[i][1]),
                min(inputs_test[i][0], inputs_test[i][1]))
               for i in range(TEST_DATA_SIZE)]

inputs_test = np.asarray(inputs_test)
labels_test = np.asarray(labels_test)

act_h = relu(np.dot(weights_h, inputs_test.T))
act_y = relu(np.dot(weights_y, act_h))

# Compute accuracy
res = act_y.T[:]
res = np.around(res)

comp = res == labels_test

succ = 0

for i in range(len(comp)):
    if comp[i][0] and comp[i][1]:
        succ += 1
    else:
        continue

print("Accuracy = {}%".format(succ / TEST_DATA_SIZE * 100))
```

Figure 6.10 Testing with even larger dataset.
Source : author

The code above tests the network with data of range 0-10.000 and results about 95% accuracy as shown below.

```
Accuracy = 95.1165%
```

Figure 6.11 Testing result.
Source : author

V. Conclusion

From the example above, we can have several conclusions. First, it is true that gradient descent doesn't always give the global minima since it was a greedy approach of the problem. Second, we prove that Artificial Neural Network is really working with gradient descent and it is using greedy approach to learn. Third, we prove that even if the Neural Network is using only 0-50 ranged dataset, the Neural Network still performs well in a dataset of 0-10000 range with about 95% accuracy.

VII. Acknowledgment

The Author thanked all the Discrete Mathematics (IF 2120) lecturers, Faza Fahleraz who taught me this concept, creators of Python and Python community, Jupyter Notebook, and NumPy developers.

References

1. A. Levitin, *Introduction to the Design & Analysis of Algorithms*, 3rd ed. Boston, Massachusetts : Pearson Education, 2012.
2. https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0, May 12, 2018
3. https://brilliant.org/wiki/backpropagation/, May 13, 2018

Statement

This project was written by me and in my own words, except for quotations from published and unpublished sources which are clearly indicated and acknowledged as such. I am conscious that the incorporation of material from other works or a paraphrase of such material without acknowledgement will be treated as plagiarism, subject to the custom and usage of the subject.

Bandung, 13 May 2018

Abram Perdanaputra - 13516083