This Toy problem is just to help you understand what your outputs look like and to sort of understand the reasoning behind any mistakes you are making. This way, you will be able to debug your problems in a more fine grained way.

Let's take a look at the network which we have constructed:
The network will be dealing with inputs which will be of 2-d vectors. The testing can be done for the following methods:

a) Verify the output of each layer after activation
b) Verify the gradients with respect to weights, biases and inputs.
c) Verify gradients for batchnorm.

We want you to use whatever code you have written to complete hw1.py file and play around to construct a neural network and verify the outputs. You can start off with a single input and then try out the example for a batch of inputs.

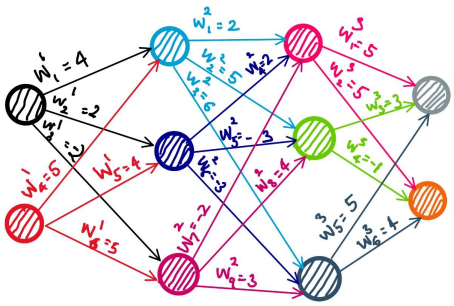
How to test?:
Steps to follow:

1. In toy.py, use the function given to generate the weights for the above architecture. The seeds are as mentioned in the file. These weights should match with the weights provided and should not change despite how many times you run the file.
2. Observe the bias initialization function.
3. Create the given input and output data.
4. Call the MLP function as mentioned in the main writeup for the given architecture.
5. Loop through the batch of inputs and perform a forward and backward pass for the same.
   (*Hint: Look at how the class methods can be called)
6. For viewing the output of forward pass for each layer, print the output of the forward method of the MLP class.
7. Create a list of gradients for weights
   (*Hint: Take advantage of the linear_layers list in hw1.py and the variables stored in the linear.py file in the backwards method.)
8. Similarly create a list of gradients for biases.
9. For viewing the gradients with respect to output of each layer, print the output of the backward function of the MLP class.

Notations used: Input layer(i/p), hidden layer (H1,H2…), Output layer (o/p)

Neural network structure:
Input layer: 2 Neurons, Hidden layers: 2 layers with 3 neurons each, Output layer: 2 Neurons.

The network will look something like this:



Weights: W1 = [[4, 2, -2],
[5, 4, 5]]
W2 = [[2, 5, 6],
[2, -3, -3],
[-2, 4, 3]]
W3 = [[5, 5],
[3, -1],
[5, 4]]

Single input, output: Input = [[4,3]], Output = [[240,4]]
Batch input: Input array = ([4,3],[5,6],[7,8]), Output array = ([240,4],[320,2],[69,5])

Case1: Activation units: ReLU( )

Neural structure with activations: [i/p (2 Neurons), ReLU( ), H1 (3 Neurons), ReLU( ), H2 (3 Neurons), ReLU( ), o/p (2 Neurons), ReLU( )]

A. Single input layerwise outputs (Post Activation) :

1. Output of hidden layer 1: [[31, 20, 7]]
2. Output of hidden layer 2: [[88, 123, 147]]
3. Output of final layer: [[1544, 905]]

Gradients:

1. Gradients with respect to Weights:

1.1. Gradient with respect to weights connecting H2 and o/p layer :
[[-53044., 13368., -16220.],
[-39783., 10026., -12165.]]

1.2. Gradient with respect to weights connecting H1 and H2 layer :
[[-37665., -22103., -37541.],
[-24300., -14260., -24220.],
[-8505., -4991., -8477.]]

1.3. Gradient with respect to weights connecting i/p and H1 layer :
[[-21032,   -352],
[-29397,   -492],
[-35133,   -588]]

2. Gradients with respect to Bias:
    2.1. Gradient with respect to biases connecting H2 and o/p layer :
    [[-13261, 3342, -4055]]
    2.2. Gradient with respect to biases connecting H1 and H2 layer :
    [[-1215, -713, -1211]]
    2.3. Gradient with respect to biases connecting i/p and H1 layer :
    [[-239., -4.]]
3. Gradients with respect to Inputs:
    3.1. Gradient with respect to output of H2 :
    [[-1215., -713., -1211.]]
    3.2. Gradient with respect to output of H1 :
    [[-13261., 3342., -4055.]]
    3.3. Gradient with respect to i/p :
    [[-38250., -73212.]]

B. Batch of input layerwise outputs (Post Activation):

1. Output of hidden layer 1: [[31, 20, 7],[50, 34, 20],[68, 46, 26]]
2. Output of hidden layer 2: [[88, 123, 147],[128, 228, 258],[176, 306, 348]]
3. Output of final layer: [[1544, 905],[2614, 1444], [3538, 1966]]

Gradients:

4. Gradients with respect to Weights:
    4.1. Gradient with respect to weights connecting H2 and o/p layer :
    [ip1: [[-53044., 13368., -16220.],
    [-39783., 10026., -12165.]],
    ip2: [[-88015., 22320., -27095.],
    [-105618., 26784., -32514.]]
    ip3: [[-27195., 6629., -8022.],
    [-31080., 7576., -9168.]]]

    4.2. Gradient with respect to weights connecting H1 and H2 layer :
    [ip1:[[-37665., -22103., -37541.],
    [-24300., -14260., -24220.],
    [-8505., -4991., -8477.]],
    ip2: [[-80250., -47750., -80150.],
    [-54570., -32470., -54502.],
    [-32100., -19100., -32060.]],

    ip3: [[-24820., -13532., -24480.],
    [-16790., -9154., -16560.],
    [-9490., -5174., -9360.]]

    4.3. Gradient with respect to weights connecting i/p and H1 layer :
    [ip1: [[-21032, -352],
    [-29397, -492],
    [-35133, -588]],

ip2: [[-40832., -256.],
[-72732., -456.],
[-82302., -516.]],
ip3: [[-11968., -880.],
[-20808., -1530.],
[-23664., -1740.]]]

5. Gradients with respect to Bias:
   5.1. Gradient with respect to biases connecting H2 and o/p layer :
   [ip1: [-13261, 3342, -4055],
   ip2: [-17603., 4464., -5419.],
   ip3: [-3885., 947., -1146.]]
   5.2. Gradient with respect to biases connecting H1 and H2 layer :
   [ip1: [-1215, -713, -1211],
   ip2: [-1605., -955., -1603.],
   ip3: [-365., -199., -360.]]
   5.3. Gradient with respect to biases connecting i/p and H1 layer :
   [ip1: [-239., -4.],
   ip2: [-319., -2.],
   ip3: [-68., -5.]]
6. Gradients with respect to Inputs:
   6.1. Gradient with respect to output of H2 :
   [ip1: [-1215., -713., -1211.],
   ip2: [-1605., -955., -1603.],
   ip3: [-365., -199., -360.]]
   6.2. Gradient with respect to output of H1 :
   [ip1: [-13261., 3342., -4055.],
   ip2: [-17603., 4464., -5419.],
   ip3: [-3885., 947., -1146.]]
   6.3. Gradient with respect to i/p :
   [ip1: [-38250., -73212.],
   ip2: [-50646., -97254.],
   ip3: [-11354., -21367.]]

Congratulations to you if you were able to verify the outputs for your network according to the given outputs. If you observe carefully, the outputs and the gradients seem pretty large and keep in mind that there was only a single forward pass and a single backward pass. Imagine if you do this 20 or 30 times, the gradients will explode. This happens because of poor weight initialization and also owing to the ReLU activation function which does not squash the output unlike Sigmoid or Tanh. You can now try the same experiment with different activation functions and for Sigmoid and Tanh, you can test out the outputs of Batchnorm too.