HW1P1 Toy Problem

1 Introduction

In this part of the assignment, we will work on a sequence of steps to build an MLP.
We will use the vector notation that we learned in class. However, in keeping with the convention used by python, our vectors will be ***row*** vectors and not ***column*** vectors as taught in class. As a result, all operations will be transposed, w.r.t. the operations in class slides.
We learned in class that a layer of perceptrons performs the following computation:

$$Y_{out} = f(Y_{in}W + B)$$

where Y_{in} is the input to the layer, W and B are the weights and bias of the layer, and $f()$ is an activation function. We also learned that the perceptron may be viewed as a sequence of *two* distinct steps:

1) Compute an affine function of the input $Z = Y_{in}W + B$
2) Apply an activation function $Y_{out} = f(Z)$

In practical implementations of neural networks, it is convenient to think of the above two steps as two distinct operations and not part of a common operation. From this perspective, the above operation is viewed as actually being a sequence of *two* layers:

1) Layer 1 computes $Y_{out1} = Y_{in}W + B$. This layer is just an affine transform and is generally referred to as a *linear* layer.
2) Layer 2 computes $Y_{out2} = f(Y_{out1})$. This layer applies an activation function to the output of the earlier linear layer and is sometimes referred to as an activation layer. The actual activation function $f()$ can be one of the various standard activation functions, such as a threshold, sigmoid, tanh, ReLU, softplus, applied individually to the components of Y_{out1} or a vector activation such as a softmax applied to the entire vector Y_{out1} to compute Y_{out2}.

Once viewed in this manner, a multi-layer perceptron (or MLP) is just a sequence of *layers* of operations on data. Each layer operates on an input vector and transforms it to produce an output vector. The actual layer may be a linear layer, an activation layer, or one of a number of other layer types we will encounter in this assignment.

2 MLP Forward

2.1 Implementing an MLP for an isolated input vector

As a preliminary exercise, we will implement a simple MLP comprising several layers, step by step. The MLP will read in an input X, and the weights and biases for the layers, and finally output a vector. In this exercise you will work with small inputs
Throughout the exercise we will use our "mytorch" framework. "mytorch" will eventually grow to a full suite of operations that you could use to build your own neural net toolkit. We will leverage object-oriented coding style in "mytorch" framework. Before continuing to read, please check the ***handout*** to know the attributes of each class in "mytorch".

2.1.1 Implementing a linear layer:

Implement a linear layer in mytorch.py. Remember that the linear layer performs the computation:

$$Y_{out} = Y_{in}W + B$$

You will write a function that takes in an input vector Y_{in}, a weights matrix W, and a bias vector B and outputs the result Y_{out}.
Dimensionality checks (using the column vector notation): If the input vector is Y_{in} N-dimensional (a $1 \times N$ vector in the row-vector notation), and your layer has M outputs (i.e M neurons),

- What is the dimension of Y_{out}?
- What is the size of W?
- What is the size of B?

Instructions on writing mytorch.linear:
In **mytorch/linear.py**, implement **Linear.forward()** function:

- Input shape (**x**): (batch size, in feature)
- Output shape (**out**): (batch size, out feature)

From the linear formula above, we can see that in actual code, the calculation of a single linear layer is composed of two steps:

1) The first step is the dot product between the input variable **x** and the weight **w**, which can be done using **np.dot** or **np.matmul**.
2) The second step is the sum of their dot product results and the bias **b**.

Hint: Add a variable to keep track of intermediate values necessary for the backward computation.

Testing the linear layer on a single input:
The Toy problem below 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.

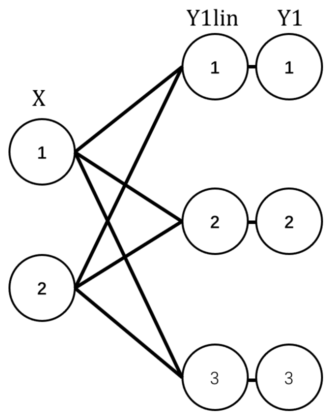

Figure 2.1a: Illustration of perceptron structure.

We will read in a single input vector and apply a linear layer to it

1) Read in the input vector X from (file name).
    - Verify that the vector is exactly **`X = [[4, 3]].`**
      To "verify" you could print out the variable (X in this case) and inspect the values visually.
2) We will now input X to a linear layer that outputs a 4-dimensional output vector.
    - What will the size of the weights matrix W be?
    - What will the size of the bias vector B be?
3) Read in the weights matrix W and bias vector B. Verify that

```
W = [[4, 2, -2],
     [5, 4, 5]]
B = [[1, 2, 3]]
```

4) Apply the linear layer to the input

```
Y1lin = linear.forward (X)
```

5) Verify that **`Y1lin = [[32, 22, 10]]`**

2.1.2 Implementing an activation layer:
Implement an activation layer in mytorch.py. Recall that an activation layer computes the operation:

$$Y_{out} = f(Y_{in})$$

You will write a function that takes in an input vector Y_{in}, applies an activation function to it, and return the result as the output vector Y_{out}. We will implement different activations within mytorch.

Instructions on writing mytorch.activation:
In **mytorch/activation.py**, implement the **forward** function for each activation class. And we also show the way to calculate the derivative during the backward pass.

- The **identity** has been implemented for you as an example.
- The output of the activation should be stored in the **state**.
- The **state** variable should be used for calculating the derivative during the backward pass.

REFERENCE: Derivative of the Sigmoid function

1) Sigmoid Forward

$$S(z) = \frac{1}{1+e^{-Z}}$$

2) Sigmoid Derivative

$$S'(z) = S(z) \cdot (1 - S(z))$$

3) Tanh Forward

$$tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-Z}}$$

4) Tanh Derivative

$$tanh'(z) = 1 - tanh(z)^2$$

5) ReLU Forward

$$R(z) = \begin{cases} z & z > 0 \\ 0 & z \le 0 \end{cases}$$

6) ReLU Derivative

$$R(z) = \begin{cases} 1 & z > 0 \\ 0 & z \le 0 \end{cases}$$

Testing the sigmoid activation on a single input:
We will read in a single input vector and apply a sigmoid activation to it.

1) Apply the sigmoid activation to the output of the linear layer.

```
Y1 = sigmoid.forward (Y1lin)
```

2) Verify that `Y1 = [[1.0, 1.0, 1.0]]`

Testing the ReLU activation on a single input:
We will read in a single input vector and apply a ReLU activation to it.

1) Apply the ReLU activation to the output of the linear layer.

```
Y1 = relu.forward(Y1lin)
```

2) Verify that `Y1 = [[32, 22, 10]]`

Testing the Tanh activation on a single input:
We will read in a single input vector and apply a Tanh activation to it.

1) Apply the Tanh activation to the output of the linear layer.

```
Y1 = tanh.forward (Y1lin)
```

2) Verify that `Y1 = [[1.0, 1.0, 1.0]]`

Testing the softmax activation on a single input:
We will apply the softmax activation to the output of the linear layer.
Question: A softmax activation outputs a probability distribution over a number of classes. If we apply a softmax to the output of e., how many classes are we modelling?

1) Apply the softmax activation to the output of the linear layer.

```
        Y1 = softmax.forward (Y1lin)
```

2) Verify that `Y1 = [[9.9995e-01, 4.5398e-05, 2.7893e-10]]`

2.1.3 Implementing a complete MLP:

We will now implement a complete MLP that nominally has two hidden layers and one output layer. The hidden layers will use a ReLU activation, and the output layer will have softmax activation.
Recall that what we call a "layer" in this context typically implies a layer of *perceptrons.* In implementation a layer of perceptrons comprises two layers – a linear layer and an activation layer. So an MLP with two hidden layers and one output layer will, in fact, have *six* layers, including three linear layers and three activation layers.

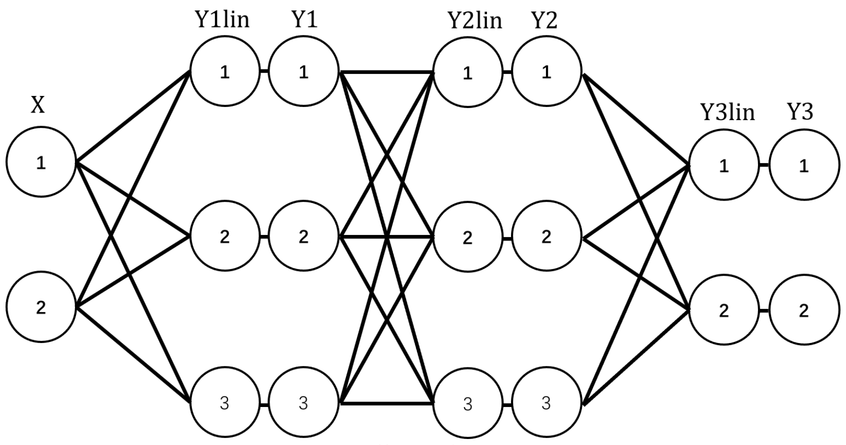

Figure 2.1b: Illustration of MLP structure.

So the complete MLP here will look like the following. Make sure all the instances have been properly initialized. We skip the initialization step in all pseudo codes in this document. Assume that input X, and the weights and bias matrices W1, W2, W3, B1, B2 and B3 have been loaded.

```
    Y1lin = linear1.forward(X)
    Y1 = relu1.forward(Y1lin)

    Y2lin = linear2.forward(Y1)
    Y2 = relu2.forward(Y2lin)

    Y3lin = linear3.forward(Y2)
    Y3 = softmax.forward(Y3lin)
```

There are other ways of writing this code of course. For instance, you can store all your layers in a list, and loop over the hidden layers. In this case it is often convenient to call your input the *zeroth* layer.

```
    Y = X
    for layer = 1:num_hidden_layers
        Ylin = LinerLayers[layer].forward(Y)
        Y = ActivationLayers[layer].forward(Ylin)
    end
```

```
# The output layer immediately follows the hidden layers
YlinFinal = LinerLayers[-1].forward(Y)
Yout = softmax.forward(Ylin)
```

We will let you figure out what the variables mean.
There are other even smarter ways of writing this. We won't get into it here.
Terminology: The outputs of the final *linear* layer (i.e. `YlinFinal`) are often called the ***logits***. You will encounter this term frequently.

Testing the MLP on a single input:
Now test your code.

1) Load the input X from X.py, and the weights and biases from W.py
   - Verify that input `X = [[4, 3]]`
   - verify that weights and biases :

```
W1 = [[ 4, 2, -2],        B1 = [[ 0, 0, 0]]
      [ 5, 4, 5]]


W2 = [[ 2, 5, 6],         B2 = [[ 0, 0, 0]]
      [ 2, -3, -3],
      [-2, 4, 3]]

W3 = [[ 5, 5],            B3 = [[ 0, 0]]
      [ 3, -1],
      [ 5, 4]]
```

2) Run your MLP on X
   - Verify that the output of the first (hidden) linear layer is
     `Y1lin = [[31, 20, 7]]`
   - Verify that the output of the first activation layer is
     `Y1 = [[31, 20, 7]]`
   - Verify that the output of the second (hidden) linear layer is
     `Y2lin = [[88, 123, 147]]`
   - Verify that the output of the second activation layer is
     `Y2 = [[88, 123, 147]]`
   - Verify that the output of the linear layer before the output activation is
     `Y3lin = [[1544, 905]]`
   - Verify that the final output is
     `Y3 = [[1544, 905]]`

2.2 Implementing an MLP for a batch of inputs:

In fact, an isolated input represented as a row vector has the shape of (1, in_feature), which is a special case of batch_size=1 compared to the batch inputs' shape (batch_size, in_feature). Thanks to Python's broadcasting mechanism, we can use our isolated input "mytorch" framework to verify the batch input results during forward propagation.

2.2.1 Implementing a linear layer:
We will read in a batch inputs and apply a linear layer to it

1) Read in the input array X from (file name).
    - Verify that the array is exactly **`X = [[4,3], [5,6], [7,8]].`**
2) With the previous linear network parameters **W** and **B**, we apply the linear layer to the input

```
W = [[ 4, 2, -2],
     [ 5, 4, 5]]


B = [[1, 2, 3]]


Y1lin = linear.forward(X)
```

3) We can verify that our batch output array

```
Y1lin = [[32, 22, 10],
         [51, 36, 23],
         [69, 48, 29]]
```

2.2.2 Implementing an activation layer:
Testing the sigmoid activation on a batch input array:
We will read in a batch input array and apply a sigmoid activation to it.

1) Apply the sigmoid activation to the output array of the linear layer.

```
Y1 = sigmoid.forward(Y1lin)
```

2) Verify that

```
Y1 = [[1.0, 1.0, 1.0],
      [1.0, 1.0, 1.0],
      [1.0, 1.0, 1.0]]
```

Testing the ReLU activation on a batch input array:
We will read in a batch input array and apply a ReLU activation to it.

1) Apply the ReLU activation to the output array of the linear layer.

```
Y1 = relu.forward(Y1lin)
```

2) Verify that

```
Y1 = [[32, 22, 10],
      [51, 36, 23],
      [69, 48, 29]]
```

Testing the Tanh activation on a batch input array:

We will read in a batch input array and apply a Tanh activation to it.

1) Apply the Tanh activation to the output array of the linear layer.

```
Y1 = tanh.forward(Y1lin)
```

2) Verify that
```
Y1 = [[1.0, 1.0, 1.0],
      [1.0, 1.0, 1.0],
      [1.0, 1.0, 1.0]]
```

Testing the softmax activation on a batch input array:
We will read in a batch input array and apply a sigmoid activation to it.

1) Apply the Tanh activation to the output array of the linear layer.

```
Y1 = softmax.forward(Y1lin)
```

2) Verify that
```
Y1 = [[9.9995e-01, 4.5398e-05, 2.7893e-10],
      [1.0000e+00, 3.0590e-07, 6.9144e-13],
      [1.0000e+00, 7.5826e-10, 4.2484e-18]]
```

2.2.3 Implementing a complete MLP:
As with the single-input complete MLP steps, let's test our code:

1) Load the input array X from X.py, and the weights and biases from W.py
    - Verify that input **X = [[4,3],[5,6],[7,8]]**
    - verify that weights and biases：

```
W1 = [[ 4, 2, -2],        B1 = [[ 0, 0, 0]]
      [ 5, 4, 5]]


W2 = [[ 2, 5, 6],         B2 = [[ 0, 0, 0]]
      [ 2, -3, -3],
      [-2, 4, 3]]

W3 = [[ 5, 5],            B3 = [[ 0, 0]]
      [ 3, -1],
      [ 5, 4]]
```

2) Run your MLP on X
    - Verify that the output of the first (hidden) linear layer is

```
Y1lin = [[31, 20, 7],
         [50, 34, 20],
         [68, 46, 26]]
```

    - Verify that the output of the first activation layer is

```
Y1 = [[31, 20, 7],
      [50, 34, 20],
      [68, 46, 26]]
```

- Verify that the output of the second (hidden) linear layer is

```
Y2lin = [[88, 123, 147],
         [128, 228, 258],
         [176, 306, 348]]
```

- Verify that the output of the second activation layer is

```
Y2 = [[88, 123, 147],
      [128, 228, 258],
      [176, 306, 348]]
```

- Verify that the output of the linear layer before the output activation is

```
Y3lin = [[1544, 905],
         [2614, 1444],
         [3538, 1966]]
```

- Verify that the final output is

```
Y3 = [[1544, 905],
      [2614, 1444],
      [3538, 1966]]
```

2.3 Computing the Loss

The loss quantifies the discrepancy between the actual output of a network, and the *target* output, and is typically computed on training inputs, for which both the input X and the target output – what we want the network output to be in response to X – are given. The loss generally takes the form of a *divergence* function (sometimes offset by an additive constant that does not affect the derivatives). Common loss functions are the cross-entropy loss and the L2 loss.

- The L2 loss is generally used when the network attempts to predict real-valued variables. For real-valued vector predictions, let Y and D be the actual and desired outputs of the network in response to an input X. The the L2 loss is given by

$$L = \frac{1}{2}\sum_i (Y_i - D_i)^2$$

  where Y_i and D_i are the i-th components Y and D of respectively.

- The cross-entropy loss is defined between two probability distributions, and is generally used for classification problems where the network outputs a probability distribution Y over the classes in response to an input. For an N-class classification network Y here would be a $1 \times N$ vector of probabilities where Y_i would be the probability assigned to the i-th class. The *target* output would be for the network to assign 0 probability for all the wrong classes, and a probability of 1 for the correct class. So D here would be a $1 \times N$ one-hot vector, where the component corresponding to the true class of the input is 1, and the rest are 0. The cross-entropy loss between Y and D is defined as

$$L = -\sum_i D_i \, log \, log \, Y_i$$

For one-hot D, this reduces to $L = -\ log\ log\ Y_c$, where Y_c is the probability assigned to the true class of the input by the network. Observe that minimizing the loss maximizes the probability of the correct class.

2.3.1 Implementing loss functions:

You are now required to implement various loss functions in **mytorch**. The loss function will take in the actual network output and the desired target output and return the loss value and will have the form

```
L = L2.forward(Y, D)
L = crossentropy.forward(Y, D)
```

Etc.

Testing the losses

We will now test your loss implementations to ensure they are working properly.

Testing your L2 loss implementation

1) Sanity check:
    - Set `Y = [[1,2,3]]` and `D = [[2,3,4]]`.
    - Compute the L2 loss `L = L2.forward(Y,D)`. Verify that the L2 loss computed is `1.0`.

2) Network check:
    - Run the network from (COMPLETE MLP SECTION) on input X.py, and consider the output Y3lin to be the final output of the network (i.e. if the network was a regression network with no final softmax).
    - Set `D = [[1540,900]]` and compute the L2 loss. Verify that the L2 loss computed is `20.5`

Testing your cross-entropy loss implementation

1) Sanity check:
    - Set `Y = [[0.2,0.3,0.5]]` and D to be a one-hot vector with target class id **c** = **2** (i.e. corresponding to the class with 0.3 probability in Y).
    - Compute the cross-entropy loss `L = crossentropy.forward(Y,D)`. Verify that the loss computed is `-log(0.3)`.
    - Repeat the above with `C = 1` and `C = 3` and verify that `L = -log(0.2)` and `-log(0.5)` respectively

2) Network check:
    - Run the network from (COMPLETE MLP SECTION) on input X.py, to obtain the final softmax output Y3.
    - Set `c = 1` and verify that the cross-entropy loss computed is `0.0`

3 MLP Backward

3.1 Computing Derivatives

To train a network, we will need to compute derivatives of the loss between the actual and target outputs of the network, with respect to its parameters. To do so however, we will also need to compute derivatives for all intermediate variables computed by the network.
Recall the derivative rules from class. By the convention we employed, the shape of the derivative of a scalar loss with respect to a vector or matrix variable is transposed with respect to the variable. Thus, the derivative for a row vector will be a column vector, and the derivative for a matrix will have the shape of its transpose, both of which make programming in python, which uses row vectors, very inconvenient.
Fortunately for us, we can redo all the arithmetic in terms of the *transposes of the derivatives*, which have the same shape as the variables themselves.
The transpose of the derivative of a scalar variable with respect to a vector is called a *gradient*, i.e. for a function such as $L = f(X)$ where L is a scalar and X is a row vector, the gradient of Y with respect to X has the same shape as X. When the output of the function is also a row vector (i.e. for a function such as $Y = f(X)$ where Y and X are both row vectors), the transpose of the derivative of Y with respect to X is the *Jacobian*. For simplicity we will (taking liberties with terminology), refer to both gradients and Jacobians generically as gradients.
All the equations below are in terms of *gradients,* rather than regular derivatives. All equations remain consistent. We will represent the gradient of a variable Y with respect to variable X as $\nabla_X Y$.
We can now specify the following rules:

- The gradient of the loss (which is a scalar) with respect to a $1 \times N$ row vector is a $1 \times N$ row vector.

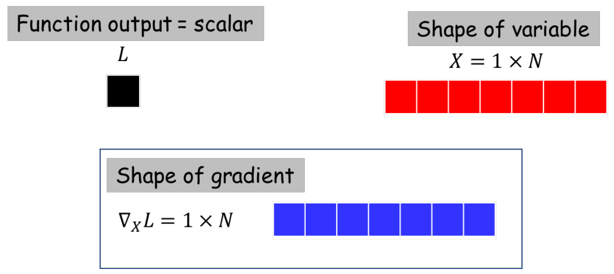

Figure 3.1a: Illustration of variable dimensions for $L = f(X)$ with scalar L and row vector X

  The i-th component of $\nabla_X L$ is the partial derivative $\frac{\partial L}{\partial x_i}$.

- The gradient of the loss with respect to an $M \times N$ matrix will also be an $M \times N$ matrix.

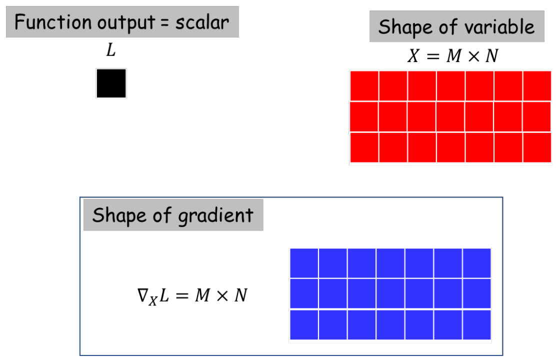

Figure 3.1b: Illustration of variable dimensions for $L = f(X)$ with scalar L and $M \times N$ matrix X

The (i, j)th element of $\nabla_X L$ is the partial derivative $\frac{\partial L}{\partial X_{i,j}}$.

- The Jacobian of a $1 \times N$ row vector with respect to a $1 \times M$ row vector will be an $N \times M$ matrix.

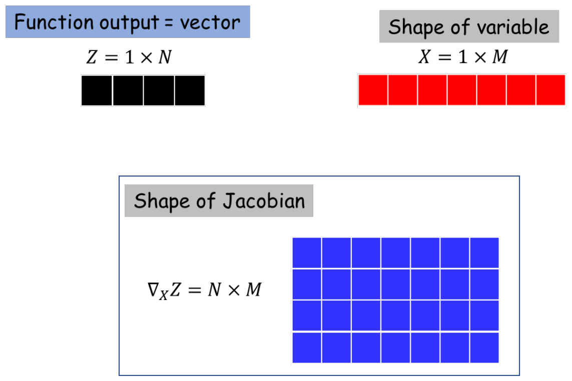

Figure 3.1c: Illustration of variable dimensions for $Z = f(X)$, where Z is an $1 \times N$ vector and X, is $1 \times M$

The (i, j)th element of $\nabla_X Z$ is the partial derivative $\frac{\partial Z_i}{\partial X_j}$. Note the indices: The numerator index in the partial derivative corresponds to the *first* (row) index in the Jacobian matrix $\nabla_X Z$, and the denominator index corresponds to the *second* (column) index.

- In any application of the chain rule, the dimensions must match up. So, for instance, if we have
  $L = f(Z)$ where $Z = h(X)$, where L is a scalar (e.g. a loss), Z is an $1 \times M$ row vector and X is an $1 \times N$ row vector, then, by the rules given just above
  - $\nabla_X L$, the gradient of L w.r.t. X must be a $1 \times N$ row vector.
  - $\nabla_Z L$, the gradient of L w.r.t. Z must be a $1 \times M$ row vector
  - $\nabla_X Z$, the Jacobian of Z w.r.t. X must be an $M \times N$ matrix
  - The chain rule for the derivatives now is given by:
$$\nabla_X L = \nabla_Z L \, \nabla_X Z$$

The following figure illustrates the chain rule relation.

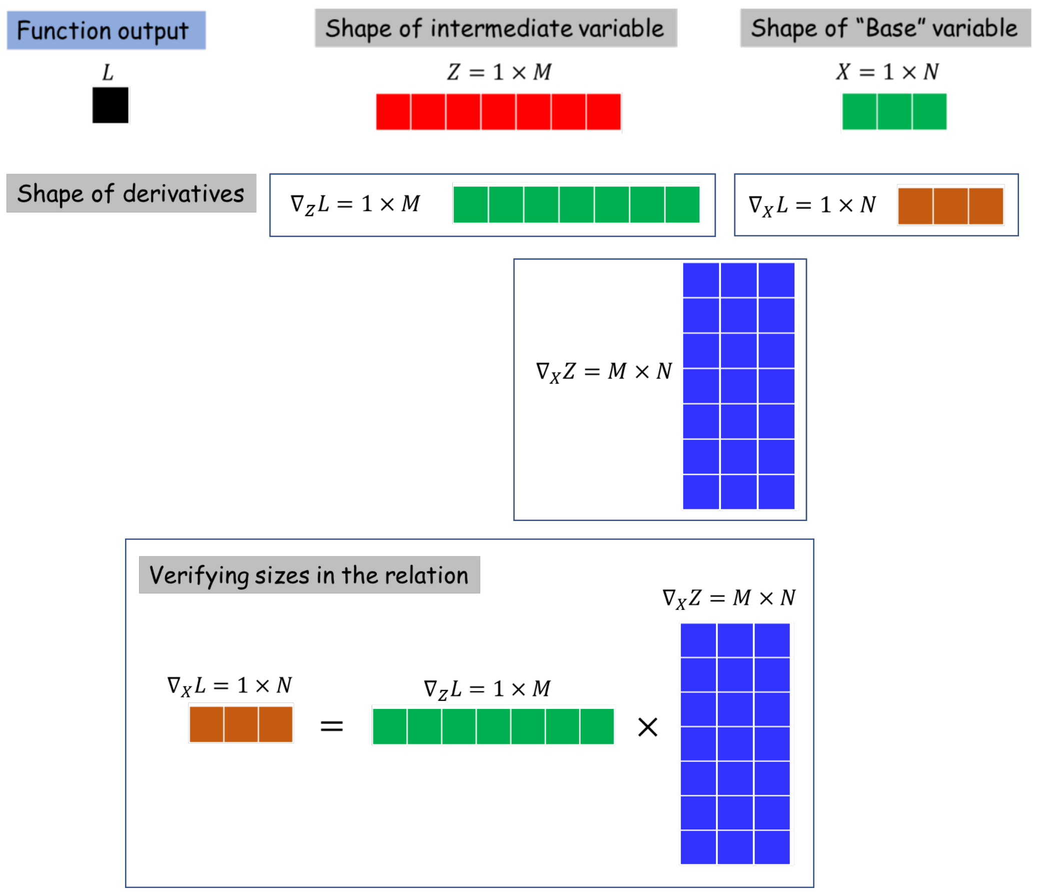
Function output
L
Shape of intermediate variable
$Z = 1 \times M$
Shape of "Base" variable
$X = 1 \times N$
Shape of derivatives
$\nabla_Z L = 1 \times M$
$\nabla_X L = 1 \times N$
$\nabla_X Z = M \times N$
Verifying sizes in the relation
$\nabla_X Z = M \times N$
$\nabla_X L = 1 \times N$
$=$
$\nabla_Z L = 1 \times M$
$\times$

Figure 3.1d: Illustration of how sizes must match up when we use the chain rule. The relation used is $L = f(Z)$ where $Z = G(X)$. L is a scalar, Z is a $1 \times M$ vector, and X is a $1 \times N$ vector. The objective is to compute $\nabla_X L$, the gradient of L w.r.t. X.

The consequence of the above rules is that the shape of the derivative of the loss with respect to *any* network parameter or intermediate variable in the network will be the *transpose* of the shape of the parameter or variable.

Additional rules:
We will additionally use the following simple rules in computing derivatives:

a. For any computation of the kind $[a, b, c] = F(d, e, f)$, where the operation takes *in* variables d, e and f, and computes values a, b and c, the derivative computation will be *backward:*

$$\partial d,\ \partial e,\ \partial f = B(\partial a, \partial b, \partial c)$$

where ∂a, ∂b, ∂c, ∂d, ∂e and ∂f are the derivatives of the loss with respect to a, b, c, d, e and f respectively and B() is the function that computes the derivative for F().

Note the reverse in the order of variables: while computing derivatives, the derivatives w.r.t. $\partial a, \partial b, \partial c$, the *output* variables a, b, c of the "forward" function $F()$ are *input* to the "backward" function $B()$. The *output* of the backward function $B()$ are $\partial d, \partial e, \partial f$, the derivatives w.r.t d, e and f, which are the inputs to $F()$.

b. In order to compute the derivatives ∂d, ∂e and ∂f, you first need ∂a, ∂b and ∂c. So, if we have a sequence of operations:

$$[a, b, c] = F(d, e, f)$$
$$[u, v, w] = G(a, b, c)$$

then we must first compute derivatives for $\partial a, \partial b, \partial c$ from $\partial u, \partial v, \partial w$,and then use those to compute $\partial d, \partial e, \partial f$. So to compute the derivatives with respect to the earliest variables in the sequence of operations, we must compute them in *reverse* order, starting with the last operation, and then working our way backwards.

Now we are set to go.

3.2 The gradient of the loss

The loss is always a scalar. It is a function of the *output* of the network and has the form:

$$L = loss(Y, D)$$

Where Y is the output of the network (in response to some training input), and D is the *desired* output in response to the same input.
As a first step of backpropagation, we require $\nabla_Y L$, the derivative of the loss with respect to the network output Y. For a simple binary classifier that outputs a single probability value between 0 and 1, or for a simple regression where the network predicts a scalar value, this derivative is just $\frac{dL}{dY}$. Please check the lectures for the formulae for this derivative for different losses for scalar outputs.
For multi-class classification or vector regression, the output Y will be a vector. In our notation, it will be a $1 \times N$ row vector, where N is the number of classes (for multi-class classification), or the dimensionality of the output prediction (for vector regression). In this case, the gradient $\nabla_Y L$ will be a $1 \times N$ row vector, whose i-th component is the partial derivative $\frac{\partial L}{\partial Y_i}$. Please refer to the lectures to see how to compute these partial derivatives for various losses, including the softmax loss.

3.2.1 A short-cut for L2 and Softmax losses

In regression problems the network does not generally have an output activation layer and ends with a linear layer. In terms of our 3-layer example earlier, the network output would simply be Y3lin, i.e. the network output Y = Y3lin. The L2 loss will generally be used in this setting.
In classification problems the network *does* generally have an output softmax layer following the final linear layer. So in our 3-layer example, the final linear layer's output is Y3lin, i.e. the logits, but the actual output of the network is Y3, i.e. Y = Y3. The cross-entropy loss will be used in this setting.
In both cases, the gradient of the loss (L2 for the regression problem, cross-entropy for the classification problem) with respect to the final *linear* layer outputs (which would be the network output for the regression problem, and the logits for the classification problem) is simply the error between actual and desired outputs.

$$\nabla_{Y3lin} L = Y - D$$

where Y is the network output and D is the desired output. For regression problems D is the actual target output. For classification problems D is the one-hot vector representation of the target output. Note the order, this is important: the gradient is Y – D and not D – Y. The logic is obvious: if Y_i is *less* than D_i, the derivative is negative, and the gradient-descent rule will

increase Y_i, which makes sense. If Y_i is greater than D_i, the derivative is positive, and the gradient-descent rule will decrease Y_i.
For proof of this simple relation, refer to the appendix. The fact that the gradient is simply the error is often why gradient back-propagation is also called error backpropagation.

3.2.2 Implementing the gradient of the loss functions:

You are now required to implement gradients for the loss functions in mytorch. We will make two distinctions:

a) For the L2 loss, we will return the loss with respect to the final network output. Here the loss will have the form
$$gradY = L2.derivative()$$
and simply return $self.Y - self.D$ (you need to store these variables when calling the forward function).

b) For the cross-entropy loss it is more convenient to return the gradient with respect to the *logits* entering the softmax, rather than the output of the softmax itself. The loss function will take in the actual network output (*not the logits)* and the desired target output and return the loss value and will have the form
$$gradZ = softmaxCrossEntropy.derivative()$$
and this too will simply return $self.Y - self.D$. (The variable is called gradZ in the code above to emphasize that it is the loss w.r.t. the *input* to the softmax)
D may be a one-hot vector, or any other target distribution. Note that although the gradient is with respect to the logits, the logits themselves are not required for the computation; we only need the output of the softmax.

Testing the loss gradient
We will now test your implementations of loss gradients.
Testing your L2 loss gradient implementation

i) Sanity check:
   a. Set `Y = [1, 2, 3]` and `D = [2, 3, 4]`.
   b. Compute the gradient of L2 loss. Verify that the L2 loss computed is `[-1, -1, -1]`.

ii) Network check:
   a. Run the network and inputs of 2.2.3, and consider the output `Y3lin` to be the final output of the network (i.e. if the network was a regression network with no final softmax).
   b. Set `D = [240, 4]` and compute the gradient of the L2 loss. Verify that the gradient computed is `[1304, 901]`.

Testing your implementation cross-entropy loss gradient at the logits

i) Sanity check:
   a. Set `Y = [0.2, 0.3, 0.5]` and `D` to be a one-hot vector with target class id `c = 2`.
   b. Compute the gradient of cross-entropy loss

```
L = softmaxCrossEntropy.derivative()
```

Verify that the gradient computed is `[0.2, -0.7, 0.5]`.

c. Repeat the above with **c = 1** and **c = 3** and verify that answers are `[-0.8, 0.3, 0.5]` and `[0.2, 0.3, -0.5]` respectively

ii) Network check:

a. Run the network and inputs of 2.2.3, to obtain the final softmax output **`Y3`**.

b. Set **`c = 1`** and verify that the gradient of the loss with respect to the logits is `[0, 0]`.

3.3 The gradient of the activation layers

The typical activation layer has the form $Y = f(Z)$, where Z is the output of the linear layer leading into the activation, and Y is the output of the activation. In principle, given $\nabla_Y L$ (the gradient of the loss w.r.t. the layer output), $\nabla_Z L$, the gradient w.r.t the input, can be computed using the chain rule as $\nabla_Z L = \nabla_Y L \; \nabla_Z Y$, where $\nabla_Z Y$ is the Jacobian of Y w.r.t. Z.
However, computing $\nabla_Z L$ in this manner requires the Jacobian $\nabla_Z Y$, which will generally be a large diagonal matrix for scalar component-wise activations. Explicitly computing and storing it and subsequently performing the matrix multiplication required by the explicit implementation of the chain rule can be memory-intensive and consequently inefficient.
Instead, we will usually directly compute $\nabla_Z L$ from $\nabla_Y L$ without explicit computation of the Jacobian.
In particular, for "scalar" activations that are applied individually to the components of Z to get the corresponding component of Y (e.g. sigmoid, ReLU, tanh), the gradient is easily computed through component-wise application of the chain rule:

$$\frac{\partial L}{\partial Z_i} = \frac{\partial L}{\partial Y_i} f^{'}(Z_i)$$

$$\nabla_Z L = \left[\frac{\partial L}{\partial Z_2}, \frac{\partial L}{\partial Z_2}, \cdots\right]$$

where $f^{'}(Z_i)$ is the derivative (or subderivative/subgradient) of the activation function computed at Z_i. For the formulae of the derivatives of specific activation functions, please refer to the lectures.

Implementing the gradient for activation layers:
You are now required to implement gradients for the activations in mytorch. You need to write functions of the kind:

```
gradZ = relu.derivative(gradY)
```

The routine takes in the loss gradient for the output of the layer and returns the loss gradient for the input vector. You must implement this for ReLu, Sigmoid and Tanh activations.

Testing the gradient of the sigmoid activation on a single input:

a. Sanity check: Set **`Z = [1, 2, 3]`** and **`gradY = [1,-1,1]`**. Assume a sigmoid activation. Compute **`gradZ`** using **`sigmoid.derivative()`**.
   - Verify that **`gradZ = [0.197, -0.105, 0.045]`**

b. From part 2.1.2, set **`Z = Y1`**. Set **`gradY = [1, -1, 1]`**.

- Verify that **`gradZ = [1.3e-14,  2.8e-10, 4.5e-5]`**

Testing the gradient of the ReLu activation on a single input:

a. Sanity check: Set **`Z = [1, 2, 3]`** and **`gradY = [1,-1,1]`**. Assume a ReLu activation. Compute **`gradZ`** using **`relu.derivative().`**
   - Verify that **`gradZ = [1, -1, 1]`**

b. From part 2.1.2, set **`Z = Y1`**. Set **`gradY = [1, -1, 1]`**.
   - Verify that **`gradZ = [1, -1, 1]`**

Testing the gradient of the Tanh activation on a single input:

a. Sanity check: Set **`Z = [1, 2, 3]`** and **`gradY = [1,-1,1]`**. Assume a Tanh activation. Compute **`gradZ`** using **`tanh.derivative()`**.
   - Verify that **`gradZ = [0.420, -0.071, 0.010]`**

b. From part 2.1.2, set **`Z = Y1`**. Set **`gradY = [1, -1, 1]`**.
   - Verify that **`gradZ = [0, 0,  8.2e-9]]`**

3.4 The gradient of a linear layer

A linear layer is the primary component of a network that has parameters – namely the weights matrix and the bias. Recall that it implements the operation $Y = ZW + B$. Thus, when we compute gradients for a linear layer, we not only compute derivatives with respect to the layer input Z, but also the parameters W (the weights) and B (the bias).
The relations are:

$$\nabla_Z L = \nabla_Y L W^\top$$
$$\nabla_W L = Z^\top \nabla_Y L$$
$$\nabla_B L = \nabla_Y L$$

Note that some of the terms in these equations are transposed with respect to the equations because we are now speaking of *gradients* rather than derivatives, and because we employ a row-vector notation for the vectors.

3.4.1 Implementing the gradient for a linear layer

You are now required to implement gradients for the linear layer in mytorch. You need to write functions of the kind:

`gradZ = linear.backward(gradY)`

The routine takes in the loss gradient for the output of the layer and returns the loss gradient for the input vector. Also, it will update the gradients of the parameters, i.e. weight and bias.

Testing the gradient of the linear layer on a single input:

a. Sanity check: Set **`Z = [1, 2]`**, **`W = [[4, 2, -2], [5, 4, 5]]`**, **`B = [1,1,1]`** and **`gradY = [1,-1,1]`**. Compute **`gradZ`**, **`gradW`** and **`gradB`** using **`linear.backward()`**.
   - Verify that **`gradZ = [0, 6]`**,
     `gradW = [[1, -1, 1], [2, -2, 2]]`,
     `gradB = [1, -1, 1]`

b. From part 2.1.1, set **`Z = X`**. Set **`gradY = [1, -1, 1]`**. Compute the gradients for the weights and biases of the first linear layer (i.e for **`W1`** and **`B1`**).
   - Verify that
     ```
     gradZ = [0, 6],
     gradW = [[ 4, -4,  4],[ 3, -3, 3]],
     gradB = [1, -1, 1]
     ```

3.5 Implementing the backward pass for a complete MLP:

We will now implement the backward pass for a complete MLP.
Specifically we will test it for the network from 2.1.3. The parameters of the network are the weights and biases W1, W2, W3, B1, B2 and B3 have been loaded.

Using your implementation of the loss, loss gradient, and the gradients of the various layers we will now

(a) compute the loss for the input, and
(b) perform a backward pass through this network.

We assume you have already loaded the input X (from X.py) and computed the forward pass through the network for 2.1.3. The *desired* target output D must also have been loaded along with X.

Your backward pass code will look like the following:

```
Loss = softmaxCrossEntropy(Y3, D)
# First compute the gradient for the input to the final softmax
layer
gradY3lin = softmaxCrossEntropy.derivative()
# Compute the gradients past the preceding linear layer.
gradY2 = linear3.backward(gradY3lin)
# The previous ReLu layer took in Y2lin and returned Y2. So…
gradY2lin = relu3.derivative()* gradY2
# The preceding linear layer took in Y1 and output Y2lin
gradY1 = linear2.backward(gradY2lin)
# The preceding ReLU layer took in Y1lin and output Y1
gradY1lin = relu2.derivative()* gradY1
# The preceding linear layer took in X and output Y1lin
gradX = linear1.backward(gradY1lin)
```

As in the case of the forward pass, there are other ways to write the above, e.g. as a loop. The loop version would look like this. Note that it exactly reverses the order of operations of the forward loop.

```
# First compute gradients for the final linear layer logits &
params
gradYlin = softmaxCrossEntropy.derivative()
gradY2 = linear3.backward(gradYlin)
```

```
# We now wind down from the last-but-one layer down to the input
for layer = num_hidden_layers-1 downto 1
    gradYlin = ActivationLayers[layer].derivative()* gradY
    gradY = LinearLayers[layer].backward(gradYlin)
End
```

Again, there are other even smarter ways of writing this. We won't get into it here.

Testing your complete MLP backward code:

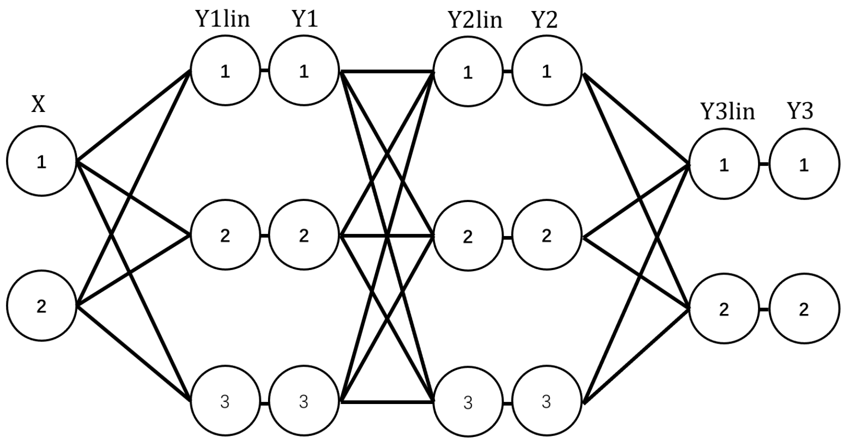

Figure 3.5a: Illustration of the MLP structure.

a. Make sure you have correctly loaded the inputs, weight and bias as part 2.1.3.
b. Verify that the gradient of the third layer are

```
gradY3lin = [[-239, -4]]
gradB3 = [[-239, -4]]
gradW3 = [[-21032, -352],
          [-29397, -492],
          [-35133, 588]]
```

c. Verify that the gradient of the output of the second layer after activation is

```
gradY2 = [[-1215, -713, -1211]]
```

d. Verify that the gradient of the third layer are

```
gradY2in = [[-1215, -713, -1211]]
gradB2 = [[-1215, -713, -1211]]
gradW2 = [[-37665, -22103, -37541],
          [-24300, -14260, -24220],
          [-8505, -4991, -8477]]
```

e. Verify that the gradient of the output of the first layer after activation is

```
gradY1 = [[-13261, 3342, -4055]]
```

f. Verify that the gradient of the third layer are

```
gradY1in = [[-13261, 3342, -4055]]
gradB1 = [[-13261, 3342, -4055]]
gradW1 = [[-53044, 13368, -16220],
           [-39783, 10026, -12165]]
```

g. Verify that the gradient of the input is

```
gradX = [[-38250, -73212]]
```

3.6 Implementing the backward pass for a complete MLP with a batch of inputs

As discussed in the lecture, we average the gradient of all parameters of the network in a batch before conducting gradient descent. To obtain the gradient of *B* (batch size) inputs, the simplest and stupidest way is just repeating the backward operation *B* times using a loop function then saving and averaging all required gradients. However, to make the computation more efficient, matrix multiplication is always involved.
Now, let us consider a simple perceptron example. We have *L=G(Y)* and *Y=ZW+B*, where *G()* is the loss function. Here, we define Z as a $BS \times N$matrix where each row represents one input vector and Y the $BS \times M$matrix where each row represents the output of the corresponding input vector. Similarly, the desired output D is also represented as a matrix with the same shape of Y.
Like what we did when batchsize=1 (Y and Z are row vectors), $\nabla_Y L$ for L2 and SoftmaxCrossEntorpyAtLogits still can be computed by

$$\nabla_Y L = Y - D$$

And the averaged gradient of $\nabla_W L$ and $\nabla_B L$ can be obtained by

$$\nabla_W L = Z^\top \nabla_Y L / BS$$
$$\nabla_B L = I \nabla_Y L / BS$$

where *I* is a $1 \times BS$ row vector which all values equal to one and *BS* is the batch size. Verify what happened in the matrix multiplication by yourself.

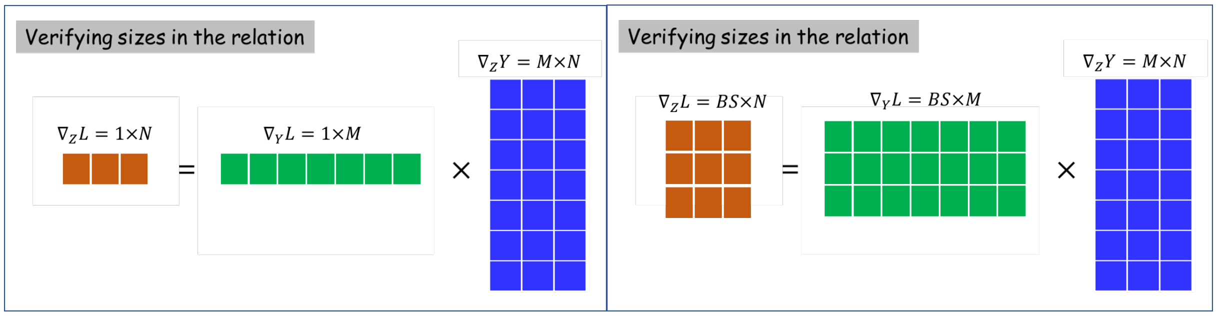
Verifying sizes in the relation
$\nabla_Z Y = M \times N$
$\nabla_Z L = 1 \times N$
$\nabla_Y L = 1 \times M$
=
×
Verifying sizes in the relation
$\nabla_Z Y = M \times N$
$\nabla_Z L = BS \times N$
$\nabla_Y L = BS \times M$
=
×

Figure 3.6a: Illustration of how sizes must match up when we use the chain rule in a batch of gradients. The left one is under the special case of BS = 1 and the right one BS=3. The relation used is $L = G(Y)$ where $Y = ZW + B$. L is a scalar, Y is a $BS \times M$ matrix, and Z is a $BS \times N$ matrix. The objective is to compute $\nabla_Z L$, the gradient of L w.r.t. Z.

To compute the gradient of $\nabla_Z L$, let us first get back to the simplest setting, batchsize=1. As shown in Figure 3.6a, we use the chain row to compute the gradient w.r.t. Z ($\nabla_Z L = \nabla_Y L \, \nabla_Z Y = \nabla_Y L W^\top$). In this equation, we found $\nabla_Z Y$ is irrelevant to Z and always equals to $W^\top$. This way, to compute a batch of gradients $\nabla_Z L$ using matrix multiplication, we can simply utilize the same equation as batchsize=1 whilst in a matrix version

$$\nabla_Z L = \nabla_Y L \, \nabla_Z Y = \nabla_Y L W^\top$$

See Figure to verify what happened in matrix multiplication. Note that $\nabla_Z Y$is not averaged in a batch.
Hints: The matrix multiplication is just leveraged for computational efficiency and the results should have no difference with those using loops. So feel free to use the loop function if you find the matrix multiplication is hard to understand at this moment.

Testing your complete MLP backward code with a batch of inputs:

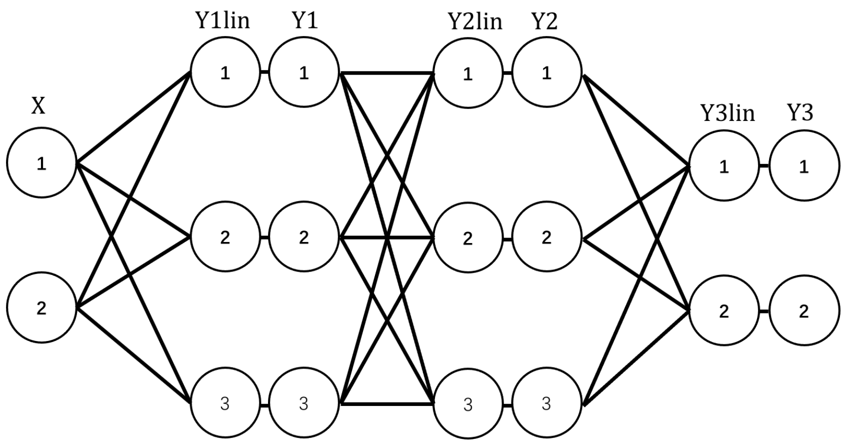

Figure 3.6b: Illustration of the MLP structure.

a. Make sure you have correctly loaded the inputs, weight and bias as part 2.2.3.

```
D = [240, 4]
```

b. Verify that the gradient of the third layer are

```
gradY3lin = [[-239, -4],
             [-319, -1],
             [-68, -4]]
gradB3 = [[-208.7,  -3.7]]
gradW3 = [[-24610.7, -496]
          [-40979, -826]
          [-47033,  -948]]
```

c. Verify that the gradient of the output of the second layer after activation is

```
gradY2 = [[-1215, -713, -1211],
          [-1605, -955, -1603],
          [-365, -199, -36]]
```

d. Verify that the gradient of the third layer are

```
gradY2in = [[-1215, -713, -1211],
          [-1605, -955, -1603],
          [-365, -199, -36]]
gradB2 = [[-1061.7,  -622.3,  -950]]
gradW2 = [[-47578.3, -27795, -47390.3]
          [-31886.7, -18628, -31760.7]
          [-16698.3,  -9755, -16632.3]]
```

e. Verify that the gradient of the output of the first layer after activation is

```
gradY1 = [[-13261, 3342, -4055],
          [-17603, 4464, -5419],
                    [-17603, 4464, -5419]]
```
f. Verify that the gradient of the third layer are
```
    gradY1in = [[-13261, 3342, -4055],
                [-17603, 4464, -5419],
                [-17603, 4464, -5419]]
    gradB1 = [[-16155.7,  4090, -4964.3]]
    gradW1 = [[-56084.7, 14105.7, -17112.3]
                [-58827, 14795.3, -17949]]
```
g. Verify that the gradient of the input is
```
    gradX = [[-38250, -73212],
              [-50646, -97254],
              [-11354, -21367]]
```
3.7 Gradient Descent

To train the MLP to produce desired outputs, we optimize the parameters of the network using gradient descent. As discussed in the lecture, gradient descent is just updating the parameters along with the opposite direction of their corresponding gradients in a fixed step size, a.k.a. learning rate. For the stochastic gradient descent, the parameters can be updated as

$$W = W - lr \times gradW$$
$$B = B - lr \times gradB$$

where lr is the learning rate.

Since the gradients calculated from single inputs are always noisy, for most of the time, we conduct gradient descent in a mini-batch. As we discussed in 3.6, the gradients are averaged in a batch before being adopted to update parameters.

Testing gradient descent on MLP with a single instance input:

We will use a learning rate of .1

a. Make sure you have correctly loaded the inputs, weight and bias as part 2.2.1.

b. Ensure that you computed all the gradients correctly (answers in 3.5)

c. Verify that the updated weights and bias matrices between the input and the first hidden layer are:
```
    W1 = [[5308.4, -1334.8,  1620 ],
     [3983.3,  -998.6,  1221.5]]
    B1 = [[1326.1, -334.2,  405.5]]
```
d. Verify that the updated weights and bias matrices between the first hidden layer and second hidden layer are:
```
    W2 = [[3768.5, 2215.3, 3760.1],
     [2432,  1423,  2419 ],
     [848.5  503.1  850.7]]
    B2 = [[121.5,  71.3, 121.1]]
```
e. Verify that the updated weights and bias matrices between the second hidden layer and output layer are:
```
    W3 = [[2108.2, 40.2],
          [2942.7, 48.2],
          [3518.3, -54.8]]
    B3 = [[23.9, 0.4]]
```

Testing gradient descent on MLP with a batch of inputs:
We will use a learning rate of .1
a. Make sure you have correctly loaded the inputs, weight and bias as part 2.2.3.
b. Ensure that you computed all the gradients correctly (answers in 3.6)
c. Verify that the updated weights and bias matrices between the input and the first hidden layer are:

```
W1 = [[5612.47, -1408.57, 1709.23],
      [5887.7, -1475.53, 1799.9 ]]
B1 = [[1615.57, -409, 496.43]]
```

d. Verify that the updated weights and bias matrices between the first hidden layer and second hidden layer are:

```
W2 = [[4759.83, 2784.5, 4745.03],
     [3190.67, 1859.8, 3173.07],
     [1667.83, 979.5, 1666.23]]
B2 = [[106.17, 62.23, 95]]
```

e. Verify that the updated weights and bias matrices between the second hidden layer and output layer are:

```
W3 = [[2466.07, 54.6 ],
     [4100.9, 81.6 ],
     [4708.3, 98.8 ]]
B3 = [[20.87, 0.37]]
```

4 Batch Normalization

Properly leveraging batch normalization is essential for NN training. As discussed in the lecture, batch normalization can speed up the training process, alleviating overfitting and mitigating gradient explosion and gradient vanishing. In this section, we will implement a batch normalization layer step by step.

4.1 Batch Normalization Training Forward

Recall that the batch normalization has two phases in forward operation: training and evaluation. We will first study the training phase.

In the training phases, batch normalization layer first norms the inputs across the batch then apply the affine transformation on the normalized data. It can be described as

$$u_B = \frac{1}{B}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{B}\sum_{i=1}^{m}(x_i - u_B)^2$$

$$\hat{x}_i = \frac{x_i - u_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma\hat{x}_i + \beta$$

where we use the annotations in the lecture.
As mentioned in the previous section, in mytorch framework, a batch of input can be denoted as a $BS \times M$matrix X where $M$is the feature dimension of inputs. In this way, to implement the BN layer in mytorch, your code should be like the following.

```
# First compute the mean of the input, mean is a 1xM row vector.
mean = np.mean(X, axis=0)
# Then compute the variance of the input, var is a 1xM row vector.
var = np.var(X, axis=0)
# Norm the input X using the mean and variance computed from above. Remember to add
a small constant to avoid zero division.
norm = (X - mean)/sqrt(var + eps)
# Do the affine transformation, gamma and beta are 1xM row vectors.
Y = norm * gamma + beta
# Update the running_mean and running_var, alpha is a scaler to control the updating
speed.
running_mean = alpha * running_mean + (1 - alpha) * mean
running_var = alpha * running_var + (1 - alpha) * var
```

Hints: You may find that dimensions of the above code are not matched. Try this in numpy to see what will happen. If you want to learn more about that, you can search numpy broadcasting on google.

Testing your complete batch norm forward code with a batch of inputs in training phase:
We will include batch norm at each of the hidden layers, updating our MLP as such (include new diagram):

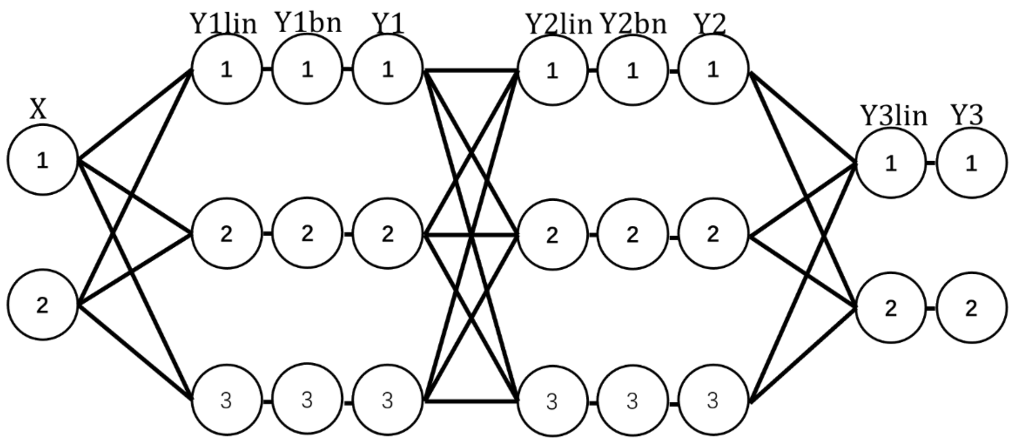

Figure 4.1b: Illustration of the MLP structure with batch normalization layer.

We will use the same weights, biases, and inputs as in 2.2.3:

```
 X = [[4,3],[5,6],[7,8]]
W1 = [[4, 2, -2],        B1 = [[ 0, 0, 0]]
      [ 5, 4, 5]]

W2 = [[ 2, 5, 6],        B2 = [[ 0, 0, 0]]
      [ 2, -3, -3],
      [-2, 4, 3]]

W3 = [[ 5, 5],           B3 = [[ 0, 0]]
      [ 3, -1],
      [ 5, 4]]
```

Also, consider the following parameters:

```
     eps = .001            alpha = .9
```

Consider M to denote the input feature size at a hidden layer. The gamma and running_var at each hidden layer will be an array of 1s of size 1*M. On the other hand, the beta and running_mean at each hidden layer will be an array of 0s of size 1*M.

a. Verify the following initial output for the first hidden layer, before batch norm:

```
Y1lin = [[31, 20, 7],
         [50, 34, 20],
         [68, 46, 26]]
```

b. Verify the following values for the first hidden layer, during batch norm:

```
mean1 = [49.67, 33.33, 17.66]

var1 = [228.22, 112.89, 62.89]

norm1 = [[-1.23562548, -1.25490605, -1.34504963],
         [0.02206474, 0.0627453, 0.29422961],
         [1.21356074, 1.19216075, 1.05082002]]

Y1bn = [[-1.23562548, -1.25490605, -1.34504963],
        [0.02206474, 0.0627453,  0.29422961],
        [1.21356074, 1.19216075, 1.05082002]]

running_mean1 = [4.96666667, 3.33333333, 1.76666667]

running_var1 = [23.72222222, 12.18888889, 7.18888889]
```

c. Verify the final output for the first hidden layer after batch norm and RELU:

```
Y1 = [[0, 0, 0],
      [0.02206474, 0.0627453, 0.29422961],
      [1.21356074, 1.19216075, 1.05082002]]
```

d. Verify the following initial output for the second hidden layer, before batch norm:

```
Y2lin = [[0, 0, 0],
         [-0.41883913, 1.09900622, 0.82684136],
         [2.70980293, 6.69460155, 6.85734227]]
```

e. Verify the following values for the second hidden layer, during batch norm:

```
mean2 = [0.7636546, 2.59786926, 2.56139454]

var2 = [1.92298436, 8.59291019, 9.34152788]

norm2 = [[-0.55054929, -0.88617988, -0.8380005 ],
         [-0.85250724, -0.51128911, -0.56748635],
         [ 1.40305653,  1.39746899,  1.40548685]]

Y2bn = [[-0.55054929, -0.88617988, -0.8380005 ],
        [-0.85250724, -0.51128911, -0.56748635],
        [ 1.40305653,  1.39746899,  1.40548685]]

running_mean2 = [0.07636546, 0.25978693, 0.25613945]

running_var2 = [1.09229844, 1.75929102, 1.83415279]
```

f. Verify the final output for the second hidden layer after batch norm and RELU:

```
Y2 = [[0, 0, 0],
      [0, 0, 0],
                    [1.40305653, 1.39746899, 1.40548685]]
```

g. Verify the following outputs for the final layer (no batchnorm):

```
Y3lin = [[0, 0],
          [0, 0],
          [18.23512387, 11.23976106]]

Y3/Softmax = [[5.00000000e-01, 5.00000000e-01],
               [5.00000000e-01, 5.00000000e-01],
               [9.99085084e-01, 9.14916212e-04]]
```

4.2 Batch Normalization Backward

In this section, you will implement the batchnorm backward function. As discussed in the lecture, we calculate the gradients of batchnorm layer as such:

$$\frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i} \cdot \gamma$$

$$\frac{\partial l}{\partial \sigma_B^2} = \sum_{i=1}^{m} \frac{\partial l}{\partial \hat{x}_i} \cdot (x_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$

$$\frac{\partial l}{\partial \mu_B} = \left(\sum_{i=1}^{m} \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}}\right) + \frac{\partial l}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_B)}{m}$$

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial l}{\partial \sigma_B^2} \cdot \frac{-2(x_i - \mu_B)}{m} + \frac{\partial l}{\partial \mu_B} \cdot \frac{1}{m}$$

$$\frac{\partial l}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i} \cdot \hat{x}_i$$

$$\frac{\partial l}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i}$$

You might currently be looking at these egyptian hieroglyphs looking calculations like:

But do not fret, we are here to help! We HIGHLY recommend that you first read through Appendix C in the HW1P1 writeup. We will shallowly go over the implementation here, but the writeup goes much more in depth.

There's a lot of calculation going on in the backward pass. Thus, we suggest breaking up the terms in your calculations. We don't want to give you all the answers, so some parts of the code will include the symbol "?" to signal that you should fill that part out yourself. Your code should look something like this:

```
#get batch size
    b = batch_size

    #we'll be using this term a lot - better make a constant!
    sqrt_var_eps = np.sqrt(self.var + self.eps)

    #Find the derivative of gamma and beta for gradient descent.
    gradGamma = np.sum(norm * gradL, axis = ???, keepdims =???)
    gradBeta = np.sum(gradL, axis = 0, keepdims = ???))

    #Find the derivative of norm
   gradNorm = gamma * gradL

    #Find the derivative of variance (this looks complicated but
isn't too bad!)
    gradVar = -.5*(np.sum((gradNorm * (x-mean))/ sqrt_var_eps
**3), axis = ???)))

    #Find the derivative of the mean. Again, looks harder than it
actually is ☺
    first_term_dmu = -(np.sum(gradNorm/sqrt_var_eps, axis
= ???)))
    second_term_dmu = - (2/b)*(gradVar)*(np.sum(x-mean, axis
= ???)))
    gradMu =  first_term_dmu + second_term_dmu
    #use all the derivative we have found to get our final
result!
    first_term_dx = gradNorm/sqrt_var_eps
    second_term_dx = gradVar * (2/b) * (x-mean)
    third_term_dx = gradMu * (1/b)

return first_term_dx + second_term_dx + third_term_dx
```

Testing your complete batch norm backward code with a batch of inputs:
Let's use the same model and values in section 4.1. Assume we just ran the forward pass of the batch data in 4.1.1. Now, we want to backpropagate with batch norm in the two hidden layers. Let's take it step by step. Assume the correct labels for our data was **[[5, 13], [5,13], [5,13]].**

a. Verify you got the following cross-entropy loss and loss gradient:

```
Cross-entropy Loss: [12.47664925, 12.47664925, 90.96139156]
```

b. Verify the gradients of the output layer:

```
gradY3lin = [[-4.5, -12.5],
             [-4.5, -12.5],
             [-4.00091492, -12.99908508]]
```

c. Verify the gradients of the weights and bias matrices between the second hidden layer and the output layer:

```
gradB3 = [[-4.33363831, -12.66636169]]

gradW3 = [[-1.87165613, -6.0810634 ],
          [-1.86382646, -6.05562458],
          [-1.87451132, -6.09034001]]
```

d. Verify the gradients of the second hidden layer:

```
gradY2 = [[-85, -1, -72.5],
          [-85, -1, -72.5],
          [-85, 0.99634034, -72.00091492]]

graddgamma2 = [[-119.2907929, 1.39243562, -101.20175003]]

graddbeta2 = [[-85, 0.99634034, -72.00091492]]

gradY2bn = [[0, 0, 0],
            [0, 0, 0],
           [-85, 0.99634034, -72.00091492]]

gradY2lin = [[4.64108825, 0.02702718, -1.39715456],
             [-4.01976864, -0.03233549,  1.5887192 ],
             [-0.62131961,  0.00530831, -0.19156464]]
```

c. Verify the gradients of the weights and bias matrices between the first and second hidden layer:

```
graddW2 = [[-0.28090203, 0.0019095, -0.06580703],
           [-0.33097962, 0.00143316, -0.04289725],
           [-0.61188154, -0.001312, 0.08871679]]

graddB2 = [[-2.96059473e-15, 1.85037171e-17, -2.96059473e-
16]]
```

d. Verify the gradients of the first hidden layer:

```
gradY1 = [[1.03438506, 13.39255864, -13.36553146],
          [1.33110046, -12.70868841, 12.67635292],
          [-2.36548552, -0.68387024, 0.68917855]]

graddgamma1 = [[-2.8412962, -1.6127009, 4.45399636]]

graddbeta1 = [[-1.03438506, -13.39255864, 13.36553146]]


gradY1bn = [[ 0,          0,          -0,         ]
           [  1.33110046, -12.70868841,  12.67635292]
           [ -2.36548552,  -0.68387024,   0.68917855]]

gradY1lin = [[-0.05464126, 0.35666977, -0.30997914],
             [0.11231814, -0.77278451, 0.98160062],
             [-0.05767688, 0.41611473, -0.67162147]]
```

e. Verify the gradients of the weights and bias matrices between the input and first hidden layer:

```
graddW1 = [[-0.0202375, 0.1585199, -0.34442127],
            [0.01619,-0.07925995, -0.13776851]]

graddB1 = [[-4.62592927e-18, -1.85037171e-17, 0]]
```

Gradient descent is similar to what we did in 3.7, except at each layer with batch norm, we also update the gamma and beta as such:

$$gamma = gamma - lr \times gradGamma$$
$$beta = beta - lr \times gradBeta$$

4.3 Batch Normalization Inference Forward

In the evaluation phase, batch normalization layer does not adopt statistics computed from the given inputs, instead, it uses the stored statistics. Let’s assume gradient descent was not performed from our backwards step, but we will still use the running mean and running var calculated.

Your code should be like the following.

# Norm the input X using the running_mean and running_var computed from the training process.

```
                norm = (X - running_mean)/sqrt(running_var + eps)
```

# Do the affine transformation, gamma and beta are 1xM row vectors.

```
                     Y = norm * gamma + beta
```

Testing your complete batch norm forward code with a batch of inputs in evaluation phase:

Consider the following input: `X = [[10,14]]`

a. Verify the following initial output for the first hidden layer, before batch norm:

```
     Y1lin = [[110, 76, 50]]
```

b. Verify the following values for the first hidden layer, during batch norm:

```
     norm1 = [[21.56500005, 20.81388639, 17.98938806]]

     Y1bn = [[21.56500005, 20.81388639, 17.98938806]]
```

c. Verify the final output for the first hidden layer after batch norm and RELU:

```
     Y1 = [[21.56500005, 20.81388639, 17.98938806]]
```

d. Verify the following initial output for the second hidden layer, before batch norm:

```
     Y2lin = [[48.77899674, 117.34089332, 120.91650531]]
```

e. Verify the following values for the second hidden layer, during batch norm:

```
     norm2 = [[46.59955049, 88.27079464, 89.09351895]]

     Y2bn = [[46.59955049 88.27079464 89.09351895]]
```

f. Verify the final output for the second hidden layer after batch norm and RELU:

```
      Y2 = [[46.59955049 88.27079464 89.09351895]]
```

g. Verify the following outputs for the final layer (no batchnorm):

```
      Y3lin = [[943.27773115, 501.10103365]]

      Y3/Softmax = [[1.00000000e+000, 9.22784414e-193]]
```