```
def initialize(self) -> None:
    #Complete this function

    '''
    initialize all biases to zero, and all weights with random sampling from a 
unifrom distribution.
    This uniform distribution should have range +/- sqrt(6 / (d_in + d_out))
    '''
    self.weights = []
    self.biases = []
    torch.manual_seed(0) #seeding

    for i in range(self.num_layers):
        inputDimension = self.layer_sizes[i]
        outputDimension = self.layer_sizes[i + 1]

            # Calculate the bound for the uniform distribution
        bound = np.sqrt(6 / (inputDimension + outputDimension))

        #Initialize weight matrix
        Weight = torch.empty(inputDimension, outputDimension).uniform_(-bound, 
bound)

        #Initialize bias  with 0
        bias = torch.zeros(outputDimension)
        self.weights.append(Weight)
        self.biases.append(bias)
    return

def forward(self, x: torch.tensor) -> torch.tensor:
    #Complete this function

    '''
    This function should loop over all layers, forward propagating the input via:
    x_i+1 = f(x_iW + b)
    Remember to STORE THE INTERMEDIATE FEATURES!
    '''
    self.features = [x]      # store input
    self.z_values = []       # clear previous z-values if any

    acti = x
    # Hidden layers: use activation function
    for i in range(self.num_layers - 1):
        z = acti @ self.weights[i] + self.biases[i]
        self.z_values.append(z)
        acti = self.activation_function.forward(z)
        self.features.append(acti)

    # Final layer: no activation function
    z = acti @ self.weights[-1] + self.biases[-1]
    self.z_values.append(z)
    self.features.append(z)
    return z

def backward(self, delta: torch.tensor) -> None:
    #Complete this function

    '''
    This function should backpropagate the provided delta through the entire MLP,
```

```
and update the weights according to the hyper-parameters
    stored in the class variables.
    '''
    i = self.num_layers - 1
    a_prev = self.features[i]  # input to final layer
    d_w = a_prev.t() @ delta
    d_b = torch.sum(delta, dim=0)

    #weights and biases using gradient descent
    self.weights[i] -= self.learning_rate * d_w
    self.biases[i] -= self.learning_rate * d_b

    # Backpropagation
    for i in reversed(range(self.num_layers - 1)):
        #multiply by the weight matrix to next layer
        delta = (delta @ self.weights[i + 1].t())

        # derivative of activation function to delta
        delta = self.activation_function.backward(delta, self.z_values[i])

        a_prev = self.features[i]
        d_w = a_prev.t() @ delta
        d_b = torch.sum(delta, dim=0)


        self.weights[i] -= self.learning_rate * d_w
        self.biases[i] -= self.learning_rate * d_b
    return
```