Problem Set 2

by Hoang Tran, modified from the version by Xide Xia and Kate Saenko.

This assignment will introduce you to:

1. Understanding the power of ReLU activation.
2. Implementing your own autograd.
3. Implementing a simple MLP.
4. Basic functionality in PyTorch

This code has been tested on Colab.

```
import torch
```

Problem 1: Universal approximation power of ReLU networks

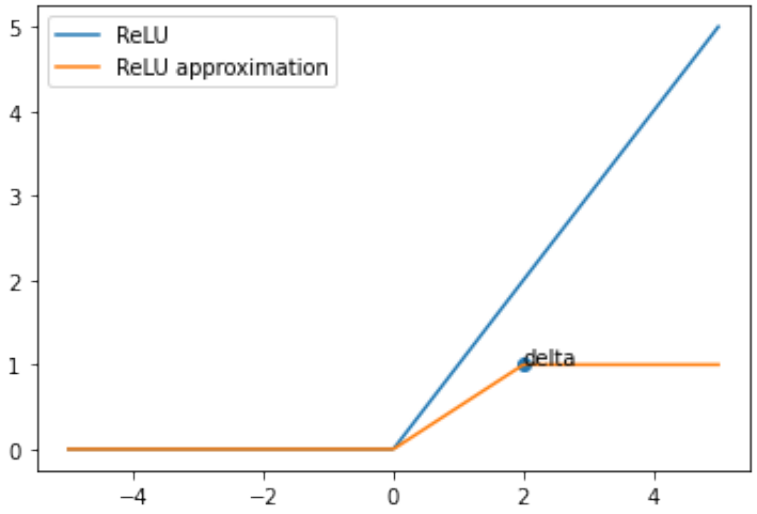
As we dicussed in class, a two layer NN with sigmoid activation function is a universal approximator, i.e: with sufficient hidden units, it can approximate any real function with desired accuracy. In this problem we want to demonstrate universal approximation power of NNs using ReLU activation units.

Q1.1

Show that, by composing only 2 hidden units in a ReLU network, i.e. $\hat{y} = \sum_{i=1}^{2} a_i \ max(0, b_i x + c_i)$, we can build an approximation to the step function $1[x > 0]$. The approximator should have value 1 for all values larger than δ and increasing linearly for any value between 0 and δ.

Solution:

```
#%%
import numpy as np
import matplotlib.pyplot as plt

#%%
def relu(x):
    return np.maximum(0, x)

#%%
def relu2(x, delta):
    return np.where(x > delta, 1, x/delta)

#%%
delta=2

#%%
x = np.linspace(-5, 5, 300)
y = relu(x)
y_approx = relu2(y, delta)

#%%
plt.plot(x, y, label="ReLU")
plt.plot(x, y_approx, label="ReLU approximation")
plt.scatter([delta],[1])
plt.annotate("delta", (delta, 1))
plt.legend()
plt.show()
```



Q1.2

Show that by composing 4 hidden units in a ReLU network; we can build an approximation to the unit impulse function of duration δ

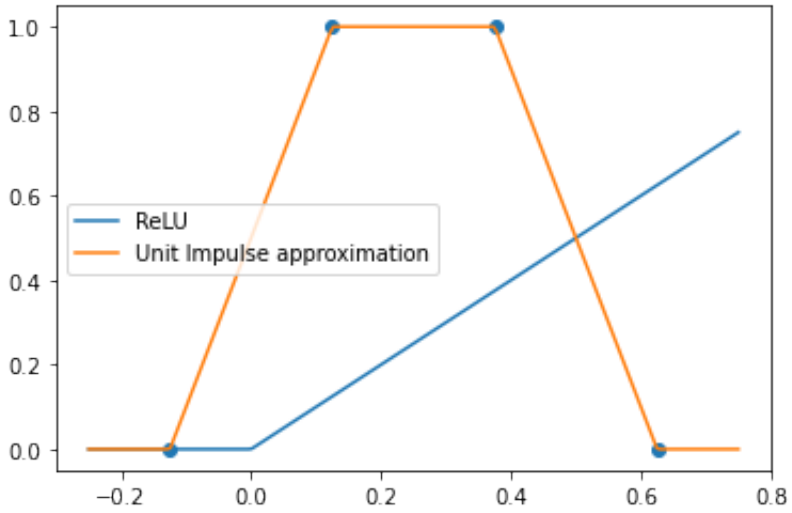
$$u_\delta(x) = 1[0 \leq x \leq \delta]$$

The approximator should have value 1 between $\frac{\delta}{4}$ and $\frac{3\delta}{4}$ and should be increasing/decreasing on either side of this for a duration of $\frac{\delta}{2}$, i.e., it should be 0 for all values less than $\frac{-\delta}{4}$ and more than $\frac{5\delta}{4}$

Solution:

```
#%%
import numpy as np
import matplotlib.pyplot as plt

#%%
delta=.5

#%%
def relu(x):
    return np.maximum(0, x)

# def relu0(x):
#     return np.maximum(0, x/2)

# def relu1(x, delta):
#     return np.where(x < (-delta / 4), 1, 0)

# def relu1(x, delta):
#     return np.where((x+delta/4) > 0, (x+delta/4) * 2/delta, x+delta/4)

# def relu2(x, delta):
#     #print(x)
#     return np.where(x> 4*delta/5, -x + 3, x)

# def relu3(x, delta):
#     return np.where(x > 1, 1, x)

# def relu4(x, delta):
#     return np.where(x < 0, 0, x)

def relu1(x, delta):
```

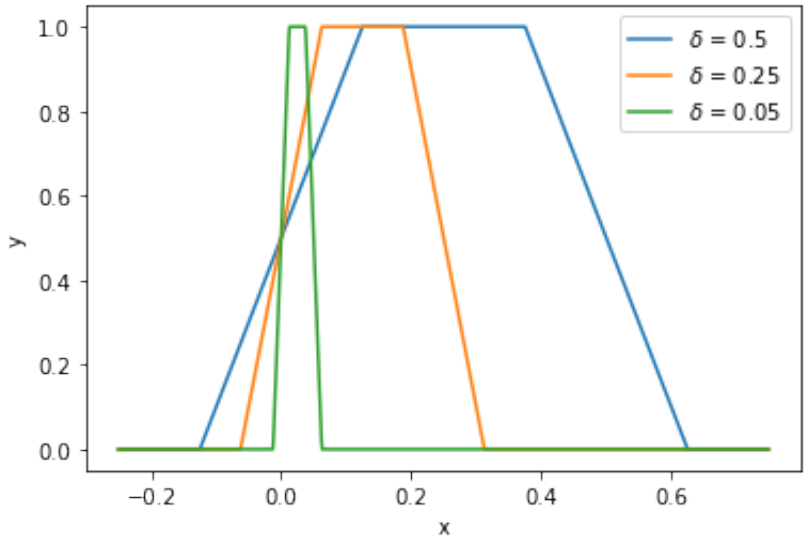
```
    #print(x)
    return np.where((x+delta/4) > 0, (x+delta/4) * 2/delta, 0)

def relu2(x, delta):
    return np.where((x-1) > 1, -(x-3), x)

def relu3(x, delta):
    return np.where(x > 1, 1, x)

def relu4(x, delta):
    return np.where(x < 0, 0, x)

# def relu1(x, delta):
#     return np.maximum(0, (x+delta/4))

# def relu2(x, delta):
#     #print(x)
#     return np.where(x > 0, x * 2/delta, x)

#%%
x = np.linspace(-0.5*delta, 1.5*delta, 200)
y = relu(x)
ya = relu1(x, delta)
ya = relu2(ya, delta)
ya = relu3(ya, delta)
ya = relu4(ya, delta)

#%%
plt.plot(x, y, label="ReLU")
plt.plot(x, ya, label="Unit Impulse approximation")
plt.scatter([-delta/4, delta/4, (3*delta)/4, (5*delta)/4],[0, 1, 1, 0])
plt.legend()
plt.show()
```



Q1.3

Using your approximator for the unit impulse function in Q1.2, complete the code given bellow to draw the approximator for different duration values δ.

```
import numpy as np
import matplotlib.pyplot as plt

def relu(x):
    return np.maximum(0,x)

def hat_u_delta(x,delta):
    return relu4(relu3((relu2((relu1(x, delta)), delta)), delta), delta)

def draw_impulse(deltas):
    delta_max = max(deltas)
    x = np.arange(-delta_max/2, 1.5*delta_max, 0.001).reshape((-1,1))
    for delta in deltas:
        plt.plot(x,hat_u_delta(x,delta),label='$\delta$ = '+str(delta))
    plt.legend();
    plt.xlabel('x')
    plt.ylabel('y');

draw_impulse([0.5, 0.25, 0.05])
# draw_impulse([1,2,4])
```



Q1.4

Imagine the idea of Riemann integral, where we approximate the integrand function with unit impulse functions -- see Figure(1)

We will approximate the function $f(x)$ defined over $[a, b]$, using N impulse functions as follows:

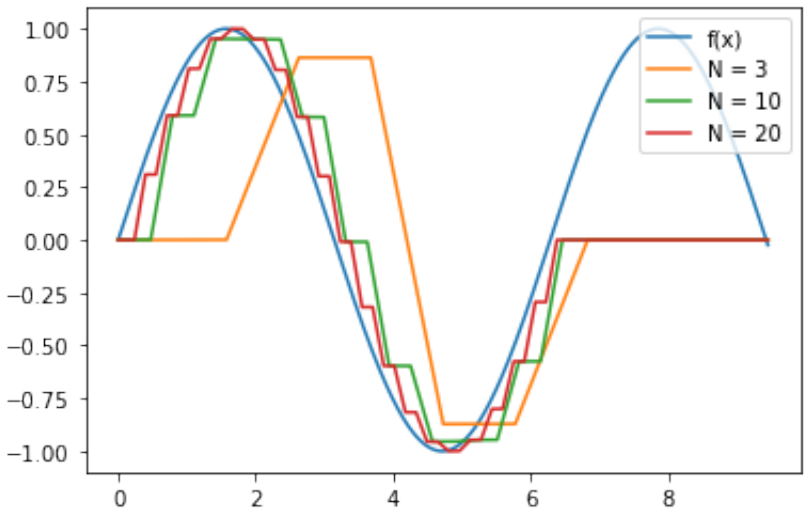
$$\hat{f}(x) = \sum_{i=0}^{N-1} f(a + i\delta)\, u_\delta(x - i\delta),$$

where:

$$\delta = \lfloor \frac{b-a}{N} \rfloor$$

Complete the code given below using the your implemented approximator in Q1.3 to approximate the $sin(x)$ function over $[0, 2\pi]$. The code will plot the approximation for different number of impule functions N.

```
def f(x):
    return np.sin(x)

def hat_f(x,N,a,b):
    delta = (b-a)/N
    total = 0
    for i in range(N):
        eff = f(a + i * delta)
        uimp = hat_u_delta((x - i* delta), delta)
        total = total + np.multiply(eff, uimp)
    return total

def draw_hat_f(N,a,b):
    x = np.arange(a, 1.5*b, 0.001).reshape((-1,1))
    plt.plot(x,f(x),label='f(x)')
    for n in N:
        y = hat_f(x,n,a,b)
        plt.plot(x,y,label=('N = '+str(n)));
    plt.legend(loc = 'upper right')

draw_hat_f([3,10,20],0,2*3.15)
```



Problem 2: Autograd implementation.

In class, we discussed the forward and back-propagation in network layers. We pass the input through a network layer and calculate the output of the layer straightforwardly. This step is called forward-propagation. Each layer also implements a function called 'backward'. Backward is responsible for the backward pass of back-propagation. The process of back-propagation follows the schemas: Input -> Forward calls -> Loss function -> derivative -> back-propagation of errors. In neural network, any layer can forward its results to many other layers, in this case, in order to do back-propagation, we sum the deltas coming from all the target layers.

In this problem, we will implement both forward and backward for the most commonly used layers including: linear, bias, ReLU, sigmoid, and mean square error.

```
'''backprop implementation with layer abstraction.
This could be made more complicated by keeping track of an actual DAG of
operations, but this way is not too hard to implement.
'''
import numpy as np


class Layer:
    '''A layer in a network.

    A layer is simply a function mapping inputs from R^n to R^d for some
    specified n and d. A neural network can usually be written as a sequence of
    layers -- eg. for input x in R^n, a 3 layer neural network might be:

    L3(L2(L1(x)))

    We can also view the loss function as itself a layer, so that the loss
    of the network is:

    Loss(L3(L2(L1(x))))

    This class is a base class used to represent different kinds of layer
    functions. We will eventually specify a neural network and its loss function
    with a list:

    [L1, L2, L3, Loss]

    where L1, L2, L3, Loss are all Layer objects.

    Each Layer object implements a function called 'forward'. forward simply
```

```
computes the output of a layer given its input. So instead of
Loss(L3(L2(L1(x))), we write
Loss.forward(L3.forward(L2.forward(L1.forward(x)))).
Doing this computation finishes the forward pass of backprop.

Each layer also implements a function called 'backward'. Backward is
responsible for the backward pass of backprop. After we have computed the
forward pass, we compute
L1.backward(L2.backward(L3.backward(Loss.backward(1))))
We give 1 as the input to Loss.backward because backward is implementing
the chain rule - it multiplies gradients together and so giving 1 as an
input makes the multiplication an identity operation.

The outputs of backward are a little subtle. Some layers may have a
parameter that specifies the function being computed by the layer. For
example, a Linear layer maintains a weight matrix, so that
Linear(x) = xW
for some matrix W.
The input to backward should be the gradient of the final loss with respect
to the output of the current layer. The output of backprop should be the
gradient of the final loss with respect to the input of the current layer,
which is just the output of the previous layer. This is why it is correct
to chain the outputs of backprop together. However, backward should ALSO
compute the gradient of the loss with respect to the current layer's
parameter and store this internally to be used in training.
'''
def __init__(self, parameter=None, name=None):
    self.name = name
    self.forward_called = False
    self.parameter = parameter
    self.grad = None

def zero_grad(self):
    self.grad = None

def forward(self, input):
    '''forward pass. Should compute layer and save relevant state
    needed for backward pass.
    Args:
        input: input to this layer.
    returns output of operation.
    '''
    return np.multiply(self.parameter, input)

def backward(self, downstream_grad):
    '''Performs backward pass.

    This function should also set self.grad to be the gradient of the final
```

```
        output of the computation with respect to the parameter.

        Args:
            downstream_grad: gradient from downstream operation in the
                computation graph. This package will only consider
                computation graphs that result in scalar outputs at the final
                node (e.g. loss function computations). As a result,
                the dimension of downstream_grad should match the dimension of
                the output of this layer.

                Formally, if this operation computes F(x), and the final
                computation computes a scalar, G(F(x)), then input_grad is
                dG/dF.
        returns:
            gradient to pass to upstream layers. If the layer computes F(x, w),
            where x is the input and w is the parameter of the layer, then
            the return value should be dF(x,w)/dx * downstream_grad. Here,
            x is in R^n, F(x, w) is in R^m, dF(x, w)/dx is a matrix in R^(n x m)
            downstream_grad is in R^m and * indicates matrix multiplication.

        We should also compute the gradient with respect to the parameter w.
        Again by chain rule, this is dF(x, w)/dw * downstream_grad
        '''




"""
Code a forward pass
"""

import numpy as np

class Layer:
    def __init__(self, parameter=None, name=None):
        self.name = name
        self.forward_called = False
        self.parameter = parameter
        self.grad = None

    def zero_grad(self):
        self.grad = None

    def forward(self, input):
        '''forward pass. Should compute layer and save relevant state
        needed for backward pass.
        Args:
```

```
            input: input to this layer.
        returns output of operation.
        '''
        raise NotImplementedError

    def backward(self, input, grad_output):
        '''backward pass. Should compute gradient with respect to input
        and parameter.
        Args:
            input: input to this layer.
            grad_output: gradient from the next layer.
        returns gradient with respect to input.
        '''
        raise NotImplementedError

    def __call__(self, input):
        self.forward_called = True
        return self.forward(input)

    def __repr__(self):
        return self.name

class Linear(Layer):
    def __init__(self, input_size, output_size, parameter=None, name='Linear'):
        super().__init__(parameter, name)
        self.input_size = input_size
        self.output_size = output_size
        if parameter is None:
            self.parameter = {'w': np.random.randn(input_size, output_size), 'b':
        else:
            self.parameter = parameter

    def forward(self, input):
        self.input = input
        return np.dot(input, self.parameter['w'
        ]) + self.parameter['b']

    def backward(self, input, grad_output):
        self.grad = {}
        self.grad['w'] = np.dot(self.input.T, grad_output)
        self.grad['b'] = np.sum(grad_output, axis=0)
        self.grad['input'] = np.dot(grad_output, self.parameter['w'].T)
        return self.grad['input']

class ReLU(Layer):
    def __init__(self, name='ReLU'):
        super().__init__(name)
```

```
    def forward(self, input):
        self.input = input
        return np.maximum(0, input)

    def backward(self, input, grad_output):
        relu_grad = input > 0
        return grad_output*relu_grad

class Sigmoid(Layer):
    def __init__(self, name='Sigmoid'):
        super().__init__(name)

    def forward(self, input):
        self.output = 1/(1 + np.exp(-input))
        return self.output

    def backward(self, input, grad_output):
        sigmoid_grad
        return grad_output*sigmoid_grad

class MSELoss(Layer):
    def __init__(self, name='MSELoss'):
        super().__init__(name)

    def forward(self, pred, target):
        self.pred = pred
        self.target = target
        return np.mean((pred - target)**2)

    def backward(self, input, grad_output):
        return 2*(self.pred - self.target)/self.pred.size

class Sequential:
    def __init__(self, layers=None, name='Sequential'):
        self.layers = layers
        self.name = name

    def add(self, layer):
        self.layers.append(layer)

    def forward(self, input):
        for layer in self.layers:
            input = layer.forward(input)
        return input

    def backward(self, input, grad_output):
        for layer in reversed(self.layers):
            grad_output = layer.backward(input, grad_output)
            input = layer.input
```

```
    def zero_grad(self):
        for layer in self.layers:
            layer.zero_grad()

    def __call__(self, input):
        return self.forward(input)

    def __repr__(self):
        return self.name

class SGD:
    def __init__(self, lr=0.001, name='SGD'):
        self.lr = lr
        self.name = name

    def update(self, layer):
        if layer.forward_called:
            layer.parameter['w'] -= self.lr*layer.grad['w']
            layer.parameter['b'] -= self.lr*layer.grad['b']

    def __call__(self, layer):
        self.update(layer)

    def __repr__(self):
        return self.name

class SGDMomentum:
    def __init__(self, lr=0.001, momentum=0.99, name='SGDMomentum'):
        self.lr = lr
        self.momentum = momentum
        self.name = name
        self.velocity = {}

    def update(self, layer):
        if layer.forward_called:
            if layer not in self.velocity:
                self.velocity[layer] = {'w': np.zeros_like(layer.parameter['w']),
            self.velocity[layer]['w'] = self.momentum*self.velocity[layer]['w'] +
            self.velocity[layer]['b'] = self.momentum*self.velocity[layer]['b'] +
            layer.parameter['w'] -= self.lr*self.velocity[layer]['w']
            layer.parameter['b'] -= self.lr*self.velocity[layer]['b']

    def __call__(self, layer):
        self.update(layer)

    def __repr__(self):
        return self.name
```

```
class Adam:
    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8, name='Adam
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.name = name
        self.t = 0
        self.m = {}
        self.v = {}

    def update(self, layer):
        if layer.forward_called:
            self.t += 1
            if layer not in self.m:
                self.m[layer] = {'w': np.zeros_like(layer.parameter['w']), 'b': n
            if layer not in self.v:
                self.v
                self.v[layer] = {'w': np.zeros_like(layer.parameter['w']), 'b': n
            self.m[layer]['w'] = self.beta1*self.m[layer]['w'] + (1-self.beta1)*l
            self.m[layer]['b'] = self.beta1*self.m[layer]['b'] + (1-self.beta1)*l
            self.v[layer]['w'] = self.beta2*self.v[layer]['w'] + (1-self.beta2)*(
            self.v[layer]['b'] = self.beta2*self.v[layer]['b'] + (1-self.beta2)*(
            m_hat = self.m[layer]['w']/(1-self.beta1**self.t)
            v_hat = self.v[layer]['w']/(1-self.beta2**self.t)
            layer.parameter['w'] -= self.lr*m_hat/(np.sqrt(v_hat)+self
            .epsilon)
            m_hat = self.m[layer]['b']/(1-self.beta1**self.t)
            v_hat = self.v[layer]['b']/(1-self.beta2**self.t)
            layer.parameter['b'] -= self.lr*m_hat/(np.sqrt(v_hat)+self
            .epsilon)

    def __call__(self, layer):
        self.update(layer)

    def __repr__(self):
        return self.name

class CrossEntropyLoss(Layer):
    def __init__(self, name='CrossEntropyLoss'):
        super().__init__(name)

    def forward(self, pred, target):
        self.pred = pred
        self.target = target
        return -np.sum(target*np.log(pred))/pred.shape[0]
```

```
    def backward(self, input, grad_output):
        return -grad_output*self.target/self.pred

class Dropout(Layer):
    def __init__(self, p=0.5, name='Dropout'):
        super().__init__(name)
        self.p = p
        self.mask = None

    def forward(self, input):
        self.mask = np.random.binomial(1, self.p, size=input.shape)
        return input*self.mask

    def backward(self, input, grad_output):
        return grad_output*self.mask
```

Below shows an example of the full implementation of the Bias layer, including the forward and backward function. Notice self.grad stores the gradient of the loss with respect to the current layer's parameter.

```
class Bias(Layer):
    '''adds a constant bias.'''

    def __init__(self, bias, name="bias"):
        super(Bias, self).__init__(np.squeeze(bias), name)
        self.weights = np.squeeze(bias)

    def forward(self, input):
        self.input = input
        return self.parameter + self.input

    def backward(self, downstream_grad):
        self.grad = np.sum(downstream_grad, tuple(range(downstream_grad.ndim - sel1
        return downstream_grad
```

Q2.1 Multiplication layers.

Let's start with the basic linear and bias layer. Show the derivatives of linear and bias layer with respect to X respectively.

$Z_{linear} = XW$

Solution (linear layer):

Assume that upstream gradient is computed to be $\frac{\partial L}{\partial Z}$,

Apply the chain rule,

$$\frac{\partial L}{\partial X} = \frac{\partial Z}{\partial X}\frac{\partial L}{\partial Z} \qquad\qquad \frac{\partial L}{\partial W} = \frac{\partial Z}{\partial W}\frac{\partial L}{\partial Z}$$

Complete the forward and backward function of the linear layer. In backward, you should ALSO set the self.grad to be the gradient of the loss with respect to the current layer's parameter.

```
class Linear(Layer):
    '''Linear layer. Parameter is NxM matrix W, input is matrix x of size B x N
    where B is batch size, output is xW.'''

    def __init__(self, weights, name="Linear"):
        super(Linear, self).__init__(weights, name)
        self.weights = weights

    def forward(self, input):
        self.input = input
        return np.dot(input, self.parameter['w']) + self.parameter['b']

    def backward(self, input, downstream_grad):
        self.grad = {}
        self.grad['w'] = np.dot(self.input.T, downstream_grad)
        self.grad['b'] = np.sum(downstream_grad, axis=0)
        self.grad['input'] = np.dot(downstream_grad, self.parameter['w'].T)
        return self.grad['input']
```

Q2.2 Activation layers.

Now let's look at the activation layers. Show the derivatives of ReLU and sigmoid.

$ReLU(x) = max(0, x)$

$\sigma(x) = \frac{1}{1+e^{-x}}$

Hint: Let's assume the gradient of ReLU is 0 when x is 0.

Solution:

Complete the forward and backward functions. There is no need to update self.grad since there is no parameter in activation layers.

```
class ReLU(Layer):
    '''ReLU layer. No parameters.'''

    def __init__(self, name="ReLU"):
        super(ReLU, self).__init__(name=name)

    def forward(self, input):
        self.input = input
        return np.maximum(0, input)

    def backward(self, input, grad_output):
        relu_grad = input > 0
        return grad_output*relu_grad



class Sigmoid(Layer):
    '''Sigmoid layer. No parameters.'''

    def __init__(self, name="Sigmoid"):
        super(Sigmoid, self).__init__(name=name)

    def forward(self, input):
        self.output = 1/(1 + np.exp(-input))
        return self.output

    def backward(self, input, grad_output):
        sigmoid_grad
        return grad_output*sigmoid_grad
```

Q2.3 Loss *layers*.

Define the mean square error as follows:

$MSE(\hat{y}) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2.$

where y is the label and $\hat{y}$ is your prediction. Show the gradient of MSE w.r.t $\hat{y}$.

Solution:

Complete the forward and backward functions.

```
class MeanSquaredError(Layer):
    '''cross entropy loss.'''

    def __init__(self, labels, name="Mean Squared Error"):
        super(MeanSquaredError, self).__init__(name="Mean Squared Error")
        self.labels = labels

    def forward(self, pred, target):
        self.pred = pred
        self.target = target
        return np.mean((pred - target)**2)

    def backward(self, input, grad_output):
        return 2*(self.pred - self.target)/self.pred.size
```

Q2.4

Now let's build a simple model using your layers, and compare the autograd results with the numeric derivatives. If everything is implemented in the correct way, the autograd results should be very close to numeric grad.

```
# This function computes the derivative numerically using the formula (f(x+delta) -
# f(x) which is the original output. Then we perturb the input by a small delta the
# the difference and divide by delta to get the derivative.
def numerical_derivative(layers, input):
    base_output = forward_layers(layers, input)
    delta = 1e-7

    for layer in layers:
        if layer.parameter is None:
            continue
        size = layer.parameter.size # total number of params
        shape = layer.parameter.shape # shape of params
        base_param = np.copy(layer.parameter)
        perturb = np.zeros(size)
        grad = np.zeros(size)

        for i in range(size):
            perturb[i] = delta # only current i-th perturb is non-zero
```

```
            layer.parameter = base_param + np.reshape(perturb, shape) # make a smal
            perturb_output = forward_layers(layers, input) # new output after addir
            grad[i] = (perturb_output - base_output) / delta # update the grad of i
            perturb[i] = 0.0 # set it back to zero

        layer.parameter = base_param
        layer.grad = np.reshape(np.copy(grad), shape)

def forward_layers(layers, input):
    '''Forward pass on all the layers. Must be called before backwards pass.'''
    output = input
    for layer in layers:
        output = layer.forward(output)
    #assert output.size == 1, "only supports computations that output a scalar!"
    return output


def backward_layers(layers):
    '''runs a backward pass on all the layers.
    after this function is finished, look at layer.grad to find the
    gradient with respect to that layer's parameter.'''
    downstream_grad = np.array([1])
    for layer in reversed(layers):
        downstream_grad = layer.backward(downstream_grad)


def zero_grad(layers):
    for layer in layers:
        layer.zero_grad()


def test_autograd():
    h = 2
    b = 3
    input = np.random.normal(np.zeros((b, h)))
    labels = np.array([0,0,1]).reshape(3,1)
    layers = [
        Linear(np.random.normal(size=(h, 2 * h))),
        Sigmoid(),
        Bias(np.array([np.random.normal()])),
        Linear(np.random.normal(size=(2 * h, 3 * h))),
        ReLU(),
        Linear(np.random.normal(size=(3 * h, 1))),
        MeanSquaredError(labels)
    ]
    output = forward_layers(layers, input)
    backward_layers(layers)
    analytics = [np.copy(layer.grad)
```

```
                for layer in layers if layer.grad is not None]
    zero_grad(layers)

    numerical_derivative(layers, input)
    numerics = [np.copy(layer.grad)
                for layer in layers if layer.grad is not None]
    # Computing the difference between the derivative of our implemented function a
    diff = np.sum([np.linalg.norm(analytic - numeric)/np.linalg.norm(numeric)
                   for analytic, numeric in zip(analytics, numerics)])

    assert diff < 1e-5, "autograd differs by {} from numeric grad!".format(diff)


if __name__ == "__main__":
    test_autograd()
    print("looking good!")
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-12-f3422e64fda6> in <module>()
     79
     80 if __name__ == "__main__":
---> 81     test_autograd()
     82     print("looking good!")

                                2 frames
<ipython-input-9-533530be9e36> in forward(self, input)
      9     def forward(self, input):
     10         self.input = input
---> 11         return np.dot(input, self.parameter['w']) +
self.parameter['b']
     12
     13     def backward(self, input, downstream_grad):

IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis
(`None`) and integer or boolean arrays are valid indices
```

Problem 3: Implementing a simple MLP.

In this problem we will develop a neural network with fully-connected layers, aka Multi-Layer Perceptron (MLP) using the layers from Problem 2. Below, we initialize toy data that we will use to develop your implementation.

```
# setup
import numpy as np
import matplotlib.pyplot as plt

# Create some toy data
X = np.linspace(-1, 1, 100).reshape(-1,1)
y = 5*X + 2 + 0.5*np.random.normal() # noisy y

print ('X = ', X.shape)
print('y = ', y.shape)
```

```
    X =  (100, 1)
    y =  (100, 1)
```

We will use the following class `TwoLayerMLP` to implement our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays.

```
class TwoLayerMLP(object):
    def __init__(self, input_size, hidden_size, label_size, std=1e-1, activation='s
        np.random.seed(0)
        self.input_size = input_size
        self.label_size = label_size

        self.params = {}

        ## TODO: Initialize your parameters below using input_size, hidden_size, la
        ## the weights of the linear layers are normally distributed with standard
        ## and mean = 0. The bias is zero. The structure of the network is as follo
        ## linear1 -> bias1 -> sigmoid -> linear 2 -> bias 2


        self.params['W1'] = ## -- ! code required
        self.params['W2'] = ## -- ! code required
        self.params['b1'] = ## -- ! code required
        self.params['b2']  = ## -- ! code required
        ##############################################################################
        #                               END OF YOUR CODE
        ##############################################################################
        self.activation = 'sigmoid'
        # Define the model
        self.models = [
                  Linear(self.params['W1']),
                  Bias(self.params['b1']),
                  Sigmoid(),
```

```
            Linear(self.params['W2']),
            Bias(self.params['b2'])
          ]

def loss(self, X, y=None, reg=0.0):
    # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    _, C = W2.shape
    N, D = X.shape

    ## TODO: Finish the forward pass, and compute the loss using the layers and
    ## layer in problem 2
    ## -- ! code required




    grads = {}
    ############################################################################
    # TODO: Compute the backward pass, computing the derivatives of the weights
    # and biases. Store the results in the grads dictionary. For example,
    # grads['W1'] should store the gradient on W1, and be a matrix of same size
    ############################################################################
    ## -- ! code required




    ############################################################################
    #                             END OF YOUR CODE
    ############################################################################
    return loss, grads

def backward_layers(self, downstream_grad):
    '''runs a backward pass on all the layers.
    after this function is finished, look at layer.grad to find the
    gradient with respect to that layer's parameter.'''
    for layer in reversed(self.models):
        downstream_grad = layer.backward(downstream_grad)

def train(self, X, y, X_val, y_val,
        learning_rate=1e-3, learning_rate_decay=0.95,
        reg=1e-5, num_epochs=10,
        batch_size=1, verbose=False):

    num_train = X.shape[0]
    iterations_per_epoch = 1
```

```
epoch_num = 0

# Use SGD to optimize the parameters in self.model
loss_history = []
grad_magnitude_history = []
train_acc_history = []
val_acc_history = []

np.random.seed(1)
for epoch in range(num_epochs):
    # fixed permutation (within this epoch) of training data
    perm = np.random.permutation(num_train)

    # go through minibatches
    for it in range(iterations_per_epoch):
        X_batch = None
        y_batch = None

        # Create a random minibatch
        idx = perm[it*batch_size:(it+1)*batch_size]
        X_batch = X[idx, :]
        y_batch = y[idx]
        # Compute loss and gradients using the current minibatch
        loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
        #print("loss", loss)
        loss_history.append(loss)

        # do gradient descent
        for param in self.params:
            self.params[param] -= grads[param] * learning_rate

        # record gradient magnitude (Frobenius) for W1
        grad_magnitude_history.append(np.linalg.norm(grads['W1']))

    # Decay learning rate
    learning_rate *= learning_rate_decay

return {
  'loss_history': loss_history,
  'grad_magnitude_history': grad_magnitude_history,
}
```

```
  File "<ipython-input-14-c01b6d3efae1>", line 15
    self.params['W1'] = ## -- ! code required
                                              ^
SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

Q3.1 Forward pass

Our 2-layer MLP uses a mean squared error loss layer defined in Problem 2.

Please take a look at method `TwoLayerMLP.loss`. This function takes in the data and weight parameters, and computes the class scores (output of the forward layer), the loss (L), and the gradients on the parameters.

- Use the layers designed in **Problem 2** and implement the first part of the function to compute `scores` and `loss`. Afterwards, run the following two test cases.

```
input_size = 1
hidden_size = 10
label_size = 1

net = TwoLayerMLP(input_size, hidden_size, label_size)
scores = forward_layers(net.models, X)
print ('(1) Your scores:\n')
print (np.linalg.norm(scores))
print ('\n')
correct_norm = 2.00385
# # The difference should be very small (< 1e-4)
print ('Difference between your scores and correct scores:')
print (np.sum(np.abs(np.linalg.norm(scores) -correct_norm)))
print ('\n')

loss, _ = net.loss(X, y, reg=0.1)
correct_loss = 5

# Since we generate random data, your loss would not be the same as the correct los
# However, the difference should be fairly small (less than 1 or 2)
print ('(2) Your loss: %f'%(loss))
print ('Difference between your loss and correct loss:')
print (np.sum(np.abs(loss - correct_loss)))
```

Q3.2 Backward pass

- Implement the second part to compute gradient of the loss with respect to the variables `W1, b1, W2,` and `b2,` stored in `grads`.

Hint: you can quickly get the gradients with respect to parameters by calling **self.backward_layers**(downstream_grad).

Now debug your backward pass using a numeric gradient check.

```
# Use numeric gradient checking to check your implementation of the backward pass
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.
def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))


def eval_numerical_gradient(f, x, verbose=True, h=0.00001):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros_like(x)
  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    oldval = x[ix]
    x[ix] = oldval + h # increment by h
    fxph = f(x) # evalute f(x + h)
    x[ix] = oldval - h
    fxmh = f(x) # evaluate f(x - h)
    x[ix] = oldval # restore

    # compute the partial derivative with centered formula
    grad[ix] = (fxph - fxmh) / (2 * h) # the slope
    if verbose:
      print (ix, grad[ix])
    it.iternext() # step to next dimension

  return grad

loss, grads = net.loss(X, y, reg=0.1)

# these should all be very small
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.1)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=F
    print ('%s max relative error: %e' % (param_name, rel_error(param_grad_num, g
```

Q3.3 Train the Sigmoid network

To train the network we will use stochastic gradient descent (SGD), implemented in `TwoLayerNet.train`. Train the two-layer network and plot the ['loss_history']. We don't expect you to optimize the training process. As long as the the loss graph looks reasonable (loss is going down), you will get full credits. All the current hyperparameters are set to 1 so feel free to play around with these values.

```
stats = net.train(X, y, X, y,learning_rate=1, reg=1e-5, batch_size = 1, num_epoch
## TODO: Plot ['loss_history'] here
## -- ! code required
```

Problem 4: Pytorch Intro

Q4.0: Pytorch tutorials

This homework will introduce you to PyTorch, currently the fastest growing deep learning library, and the one we will use in this course.

Before starting the homework, please go over these introductory tutorials on the PyTorch webpage:

- 60-minute Blitz

```
import torch
```

The `torch.Tensor` class is the basic building block in PyTorch and is used to hold data and parameters. The `autograd` package provides automatic differentiation for all operations on Tensors. After reading about Autograd in the tutorials above, we will implement a few simple examples of what Autograd can do.

Q4.1. Simple function

Use `autograd` to do backpropagation on a simple function, $f = (x + y) * z$.

Q4.1.1 Create three inputs with values $x = -2$, $y = 5$ and $z = -4$ as tensors and set `requires_grad=True` to track computation on them.

```
## -- ! code required

x = torch.tensor([-2.], requires_grad=True)
y = torch.tensor([5.], requires_grad=True)
z = torch.tensor([-4.], requires_grad=True)
f = (x + y) * z

print(f.backward())
```

```
    None
```

Q4.1.2 Compute the $q = x + y$ and $f = q \times z$ functions, creating tensors for them in the process. Print out q, f, then run `f.backward(retain_graph=True)`, to compute the gradients w.r.t. x, y, z. The `retain_graph` attribute tells autograd to keep the computation graph around after the backward pass as opposed deleting it (freeing some memory). Print the gradients. Note that the gradient for q will be `None` since it is an intermediate node, even though `requires_grad` for it is automatically set to `True`. To access gradients for intermediate nodes in PyTorch you can use hooks as mentioned in this answer. Compute the values by hand to verify your solution.

```
## -- ! code required
```

Q4.1.3 If we now run `backward()` again, it will add the gradients to their previous values. Try it by running the above cell multiple times. This is useful in some cases, but if we just wanted to re-compute the gradients again, we need to zero them first, then run `backward()`. Add this step, then try running the backward function multiple times to make sure the answer is the same each time!

```
## -- ! code required
```

Q4.2 Neuron

Implement the function corresponding to one neuron (logistic regression unit) that we saw in the lecture and compute the gradient w.r.t. x and w. The function is $f = \sigma(w^T x)$ where $\sigma()$ is the sigmoid function. Initialize $x = [-1, -2, 1]$ and the weights to $w = [2, -3, -3]$ where w_3 is the bias. Print out the gradients and double check their values by hand.

```
## -- ! code required
```

Q4.3. torch.nn

We will now implement the same neuron function f with the same variable values as in Q4.2, but using the `Linear` class from `torch.nn`, followed by the Sigmoid class. In general, many useful functions are already implemented for us in this package. Compute the gradients $\partial f / \partial w$ by running `backward()` and print them out (they will be stored in the Linear variable, e.g. in `.weight.grad`.)

```
## -- ! code required
```

Q4.4 Module

Now lets put these two functions (Linear and Sigmoid) together into a "module". Read the Neural Networks tutorial if you have not already.

Q4.4.1 Make a subclass of the `Module` class, called `Neuron`. Set variables to the same values as above. You will need to define the `__init__` and `forward` methods. Our neuron would have 1 linear layer and 1 sigmoid layer.

```
import torch.nn as nn

class Neuron(nn.Module):

    def __init__(self):
        super(Neuron, self).__init__()
       ## -- ! code required

    def forward(self, x):
       ## -- ! code required
        return x
```

Q4.4.2 Now create a variable of your `Neuron` class called `my_neuron` and run backpropagation on it. Print out the gradients again. Make sure you zero out the gradients first, by calling `.zero_grad()` function of the parent class. Even if you will not re-compute the backprop, it is good practice to do this every time to avoid accumulating gradient!

```
## -- ! code required
```

Q4.5. Loss and SGD

Now, lets train our neuron on some data. The code below creates a toy dataset containing a few inputs x and outputs y (a binary 0/1 label), as well as a function that plots the data and current solution.

```
import matplotlib.pyplot as plt

# create some toy 2-D datapoints with binary (0/1) labels
x = torch.tensor([[1.2, 1], [0.2, 1.4], [0.5, 0.5],
                  [-1.5, -1.3], [0.2, -1.4], [-0.7, -0.5]])
y = torch.tensor([0, 0, 0, 1, 1, 1 ])

def plot_soln(x, y, params):
  plt.plot(x[y==1,0], x[y==1,1], 'r+')
  plt.plot(x[y==0,0], x[y==0,1], 'b.')
  plt.grid(True)
  plt.axis([-2, 2, -2, 2])

  # NOTE : This may depend on how you implement Neuron.
  #   Change accordingly
  w0 = params[0][0][0].item()
  w1 = params[0][0][1].item()
  bias = params[1][0].item()

  print("w0 =", w0, "w1 =", w1, "bias =", bias)
  dbx = torch.tensor([-2, 2])
  dby = -(1/w1)*(w0*dbx + bias)  # plot the line corresponding to the weights and b
  plt.plot(dbx, dby)

params = list(my_neuron.parameters())
plot_soln(x, y, params)
```

Q4.5.1 Declare an object `criterion` of type `nn.CrossEntropyLoss`. Note that this can be called as a function on two tensors, one representing the network outputs and the other, the targets that the network is being trained to predict, to return the loss. Print the value of the loss on the dataset using the initial weights and bias defined above in Q4.2.

```
## -- ! code required
```

Q4.5.2 Print out the chain of `grad_fn` functions backwards starting from `loss.grad_fn` to demonstrate what backpropagation will be run on.

```
## -- ! code required
```

Q4.5.3 Run the Stochastic Gradient Descent (SGD) optimizer from the `torch.optim` package to train your classifier on the toy dataset. Use the entire dataset in each batch. Use a learning rate of 0.01 (no other hyperparameters). You will need to write a training loop that uses the `.step()` function of the optimizer. Plot the solution and print the loss after 1000 iterations.

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(my_neuron.parameters(), lr=0.01)

# training loop
for i in range(10000):
  ## -- ! code required




print("loss =", loss.item())
params = list(my_neuron.parameters())
plot_soln(x, y, params)
```

Q4.5.4 How many thousands of iterations does it take (approximately) until the neuron learns to classify the data correctly?

Solution

Q4.6. Hidden space ablation

Now let's look at the size of network's hidden space. We will create and train a **2-layer MLP** network on the SVHN Dataset.

The SVHN dataset consists of photos of house numbers, collected automatically using Google's Street View. Recognizing multi-digit numbers in photographs captured at street level is an important component of modern-day map making. Google's Street View imagery contains hundreds of millions of geo-located 360 degree panoramic images. The ability to automatically transcribe an address number from a geo-located patch of pixels and associate the transcribed number with a known street address helps pinpoint, with a high degree of accuracy, the location of the building it represents. Below are example images from the dataset. Note that for this dataset, each image (32x32 pixels) has been cropped around a single number in its center, which is the number we want to classify.

In this problem, we turn the input images into grayscale and then flat them into 1-D vector. First, download the SVHN dataset using `torchvision` and display the images in the first batch. Take a look at the Training a Classifier tutorial for an example. Follow the settings used there, such as the normalization, batch size of 4 for the `torch.utils.data.DataLoader`, etc.

```
# solution here
import torch
import torchvision
import torchvision.transforms as transforms

import matplotlib.pyplot as plt
import numpy as np

# functions to show an image
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))

transform = transforms.Compose(
    [transforms.Grayscale(),
     transforms.ToTensor(),
     transforms.Normalize((0.5), (0.5))])

trainset = torchvision.datasets.SVHN(root='./data', split='train', transform=transf
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num

classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')



# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

Q4.6.1 2-layer MLP

Next, we will train a 2-layer MLP on the data. We have defined a simple 2-layer MLP for you with two fc layers and ReLU activation.

```
import torch.nn as nn
import torch.nn.functional as F

class Neuron(nn.Module):
    def __init__(self, hidden_size):
        super(Neuron, self).__init__()
        self.l1 = nn.Linear(1024, hidden_size)
        self.l2 = nn.Linear(hidden_size, 10)

    def forward(self, x):
        x = x.view(-1, 1024)
        x = F.relu(self.l1(x))
        x = self.l2(x)
        return x
```

You can check the number of parameters in the model by printing out the model summary.

```
def model_summary(model):
  print("model_summary")
  print()
  print("Layer_name"+"\t"*7+"Number of Parameters")
  print("="*100)
  model_parameters = [layer for layer in model.parameters() if layer.requires_grad]
  layer_name = [child for child in model.children()]
  j = 0
  total_params = 0
  print("\t"*10)
  for i in layer_name:
    print()
    param = 0
    try:
      bias = (i.bias is not None)
    except:
      bias = False
    if not bias:
      param =model_parameters[j].numel()+model_parameters[j+1].numel()
      j = j+2
    else:
      param =model_parameters[j].numel()
      j = j+1
    print(str(i)+"\t"*3+str(param))
    total_params+=param
  print("="*100)
  print(f"Total Params:{total_params}")

my_neuron = Neuron(10)
model_summary(my_neuron)
```

Instantiate the cross-entropy loss `criterion`, and an SGD optimizer from the `torch.optim` package with learning rate $.001$ and momentum $.9$. You may also want to enable GPU training using `torch.device()`.

```
import torch.optim as optim

## -- ! code required
criterion = None
optimizer = None

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# if we set the hardware to GPU in the Notebook settings, this should print a CUDA
print(device)

my_neuron.to(device)
```

Q4.6.2 Training

Complete the training loop that makes five full passes through the dataset (five epochs) using SGD. Your batch size should be 4 and hidden size is 10.

```
my_neuron = Neuron(10)

# num of epoch
stats = []
for epoch in range(5):
  running_loss = 0.0
  for i, data in enumerate(trainloader, 0):
    inputs, labels = data
    inputs, labels = inputs.to(device), labels.to(device)

    ## -- ! code required



    # print statistics
    running_loss += loss.item()
    if i % 2000 == 1999:    # print every 2000 mini-batches
        print('[%d, %5d] loss: %.3f' %
              (epoch + 1, i + 1, running_loss / 2000))
        stats.append(running_loss / 2000)
        running_loss = 0.0
```

Train the model again but this time set the hidden size as 100.

```
my_neuron_large = Neuron(100)

# create your optimizer
optimizer = ## -- ! code required

# num of epoch
stats_v2 = []
for epoch in range(5):
  running_loss = 0.0
  for i, data in enumerate(trainloader, 0):
    inputs, labels = data
    #inputs, labels = inputs.to(device), labels.to(device)
    optimizer.zero_grad()   # zero the gradient buffers
    outputs = my_neuron_large(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    # print statistics
    running_loss += loss.item()
    if i % 2000 == 1999:    # print every 2000 mini-batches
        print('[%d, %5d] loss: %.3f' %
              (epoch + 1, i + 1, running_loss / 2000))
        stats_v2.append(running_loss / 2000)
        running_loss = 0.0
```

Q4.6.3

Plot the loss of the model with hidden_size=10 with hidden_size=100 and compare the performances.

```
## -- ! code required
```

Solution here:

✓ 0s completed at 2:42 PM