HW 1: Math Foundation and Programming

This assignment intends to:

- Test your Python programming skills
- Understand gradients and backpropagation
- Think classical regression problems with a deep learning mind

- Suppose you have a model $\hat{y} = \sigma(z)$, where:
  - $ z= w^T x + b,~i.e.~z=w_1*x_1+w_2*x_2+w_3*x_3+ w_4*x_4 + b$,
  - σ is the sigmoid function, i.e. $\sigma(z) = \frac{1}{(1+e^{-z})}$, and
  - w, b are parameters. b is a scalar, $x,w~\in R^4$, specifically, $w = [w_1, w_2, w_3, w_4]^T$, $x = [x_1, x_2, x_3, x_4]^T$.
- Your ground truth lable $y=0~or~1$. With a sample (x, y), You measure your model performance by two possible cost functions:
  - Squared error: $L=\frac{1}{2}(y-\hat{y})^2$
  - Cross entropy: $L=-[y*\ln{\hat{y}}+(1-y)*ln{(1-\hat{y})}]$

Following the instruction below to program your solution in Python notebook step by step carefully:

Q1. Write a function to calculate each of the following partial derivatives. The inputs to the function can be all the variables in the model and the returned derivatives are expressions of these variables. An example is given below.

- `g_L_2_z(L, z, y, yhat, func)` : function to calculate $ \frac{\partial{L}}{\partial{z}}$. `func` is the name of the loss function
- `g_z_2_w(z, x, w )` : calculate $ \frac{\partial{z}}{\partial{w}}$
- `g_z_2_b(z, b )` : calculate $ \frac{\partial{z}}{\partial{b}}$

Note, these gradients are very simple. You really don't have to use gradient packages such as PyTorch.autograd. Just define each gradient as an expression of input variables.

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

In [2]:
```
def g_z_2_b():

    return 1

def g_z_2_w(x):

    # add your code here


def g_L_2_z(y, yhat, func):

    if func=='CrossEntropy':

        # add your code here

    else:

        # add your code here
```

Q2. Write a forward pass function `forward(x, w, b, func)` to calculate variables $z, \hat{y}, L$, with given $x, y, w, b$

In [3]:
```
def forward(x, w, b, func):

    z, yhat, L = None, None, None

    # add your code here

    return z, yhat, L
```

Q3. Write a function `gradient_desc (v, g, lr)` to adjust a parameter value $v$ by its gradient $g$, i.e. return the new value of parameter $v$ as $v$ $\leftarrow$ $v-lr*g$, where $lr$ is the learning rate.

In [4]:
```
def gradient_desc(v, dev, lr):

    # add your code here
```

Q4. Write a function `train(x, y, w_0, b_0, func, lr, n)` as follows:

1. Initialize $w$, $b$ with w_0, b_0
2. Use a loop of $n$ rounds to do the following
   A. Call the forward function you defined in Q2 to calculate $z, \hat{y}, L$
   B. Apply backpropagation to calculate the partial derivatives $\frac{\partial{L}}{\partial{w}}, \frac{\partial{L}}{\partial{b}}$ using the functions you defined in Q1.
   C. Update $w, b$ using the function `gradient_desc` you defined in Q3
   D. record $\hat{y}$, $L$
3. Return the history of $\hat{y}$, and $L$

In [5]:
```python
def train(x, y, w0, b0, func, lr, n):

    Yhat, C = None, None

    # add your code here

    return Yhat, C
```

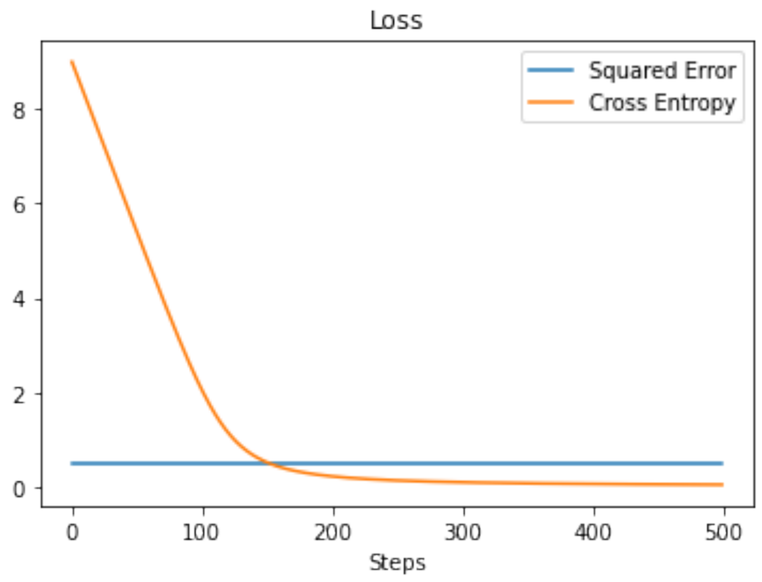
Q5. Test your program with these two test cases and plot the history of loss $L$ (i.e. learning curves) under different loss functions. An example plot for case A has been given.

Case A: $x=[1.0,0.5,-1.0, -2.0]^T, y=1, w_0=[-2,-2,1,2]^T, b_0=-1, lr = 0.01$

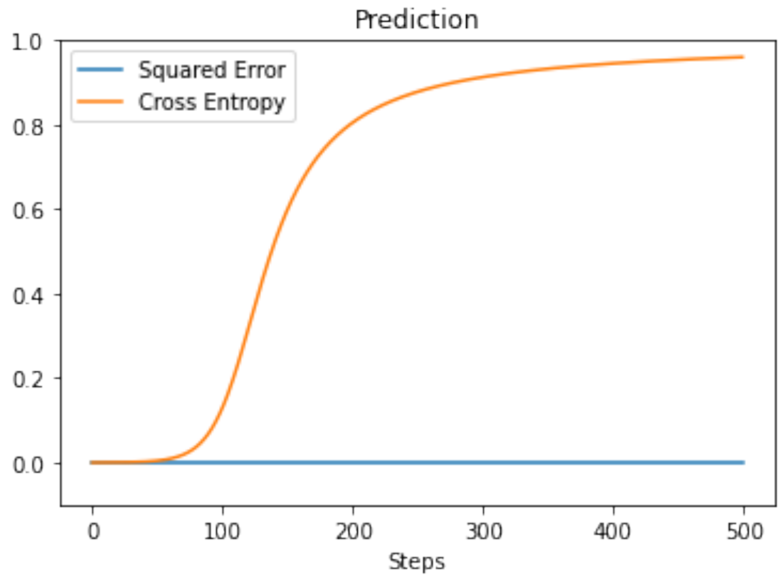
In [7]:
```python
# Case A:

x=np.array([1.0,0.5,-1.0, -2.0])
y=1
w_0=np.array([-2,-2,1,2])
b_0=-1
lr = 0.01
n = 500
```

In [8]:
```python
# Add your code here
```

```
Out[8]: [<matplotlib.lines.Line2D at 0x114906be0>]

Out[8]: [<matplotlib.lines.Line2D at 0x114906fa0>]

Out[8]: Text(0.5, 1.0, 'Loss')

Out[8]: Text(0.5, 0, 'Steps')

Out[8]: <matplotlib.legend.Legend at 0x114906e20>
```



```
Out[8]: [<matplotlib.lines.Line2D at 0x114a1ca60>]

Out[8]: [<matplotlib.lines.Line2D at 0x114a1ccd0>]

Out[8]: <matplotlib.legend.Legend at 0x114a0cac0>

Out[8]: Text(0.5, 0, 'Steps')

Out[8]: (-0.1, 1.0)

Out[8]: Text(0.5, 1.0, 'Prediction')
```



Case B: $x=[-1.0,-0.5,-1.0, -2.0]^T, y=1, w_0=[-2,-2,1,-2]^T, b_0=-1, lr = 0.01$

In [9]:
```
# Case B:

x=np.array([-1.0,-0.5,-1.0, -2.0])
y=0
w_0=np.array([-2,-2,1,-2])
b_0=-1
lr = 0.01
n = 500
```

In [ ]:
```
# add your code here
```

Q6. Carefully observe the learning curves under different loss functions. One difference you can find is the learning curves with Squared Error loss function are flat, shown little progress, and the prediction is alway far away from $y$. Can you explain the differences between these two curves in each plot? Write down your analysis below as markdowns.

Q7 (Bonus). In this experiment, we find that loss functions can have a big impact on the learning curves. Can you enumerate other two decisions or strategies that can improve the learning curves? Implement these strategies in this model and demonstrate their effects clearly.

In [ ]:
```
```