Q1 True/False Questions

1.1

False. For instance, l1 need only one final pruning step with a small constant threshold (e.g. 1e-4) is needed to reach a sparse model, can reach similar sparse level as the iterative pruning.

1.2

True. When we use hard thresholding the sparsity will keep the same while soft thresholding like proximal l1, the sparsity is gradually increasing.

1.3

False. It cannot solve biased problem: the absolute value of large element always shrinks by $\lambda\lambda$, leading to loss of variance

1.4

False. We should apply L2 norm within each group and apply L1 between groups.

1.5

False. The Hoyer-Square has the same range and a similar minima structure as the l0 norm, can be considered as a continuous approximation of the l0

1.6

True. The added proximity term will allow smootherconvergence of the overall objective.

1.7

True. The objective of the trimmed l1 is proven to be equivalent to sparse minimization, which is much stronger than the guarantee on Lasso.

1.8

False. some steps are regularized neural network training problem, which is costly to reach an acceptably optimal solution. Result could suffer if this step is not optimal enough. Besides, it requires two-times more variables during the optimization

1.9

False. STE train with full precision weight w, loss computation with quantized w′.

1.10

True. Perform mixed-precision quantization (assign different precisions to different layers) can provide better size/latency-accuracy tradeoff than fixed quantization

Lab1

In [1]:
```
from lab1 import *
X =np.array([[1,-2,-1,-1,1],[2, -1, 2, 0, -2],[-1, 0, 2, 2, 1]])
Y = np.array([-7,-1,-1])
W0 = [0,0,0,0,0]
u = 0.02
epochs = 200
```

Lab1 (a)

$\nabla L = 2\sum_i (X_i W - y_i) X_i^T, i \in \{1,2,3\}$

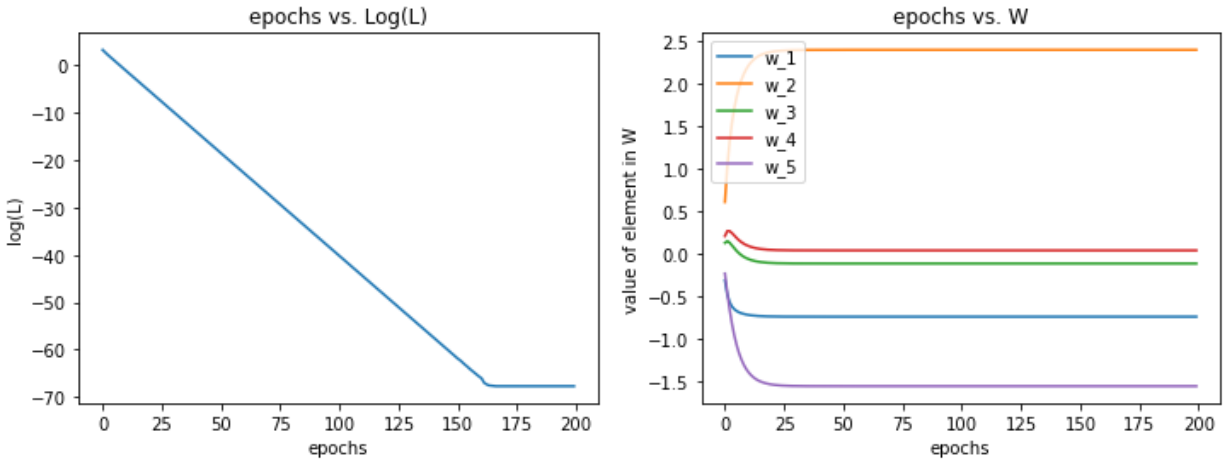
$W^{k+1} = W^k - 2u\sum_i (X_i W^k - y_i) X_i^T, i \in \{1,2,3\}$

Lab1 (b)

From the figure we can see W is converging to an optimal solution where the loss is very small.

W is not a sparse solution as most of elements of W is non-zero.

In [2]:
```
do_b(X,Y,W0,u,epochs)
```

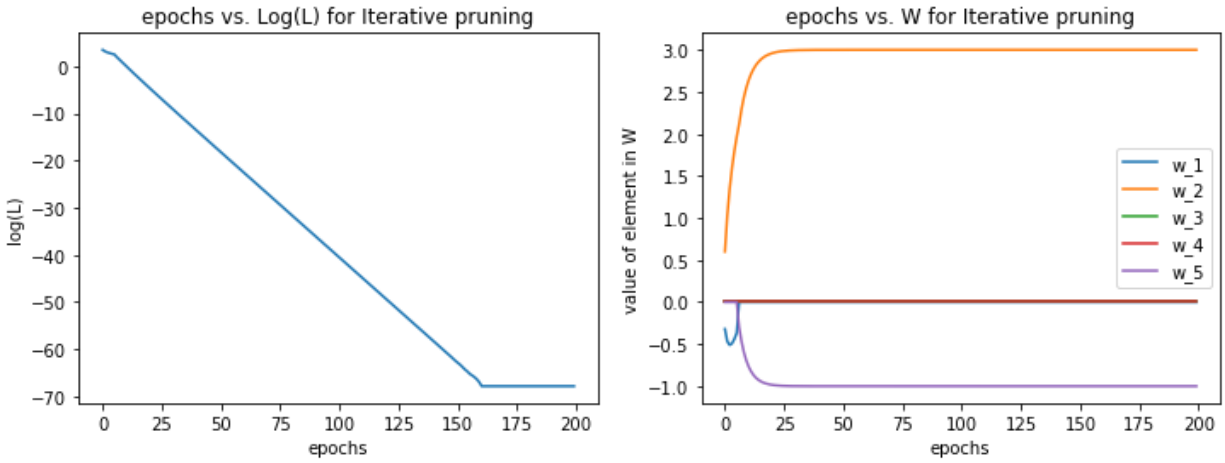


Lab1 (c)

Yes. W is converging to an optimal solution.

W is converging to a sparse solution where only two elements of W are non-zero.

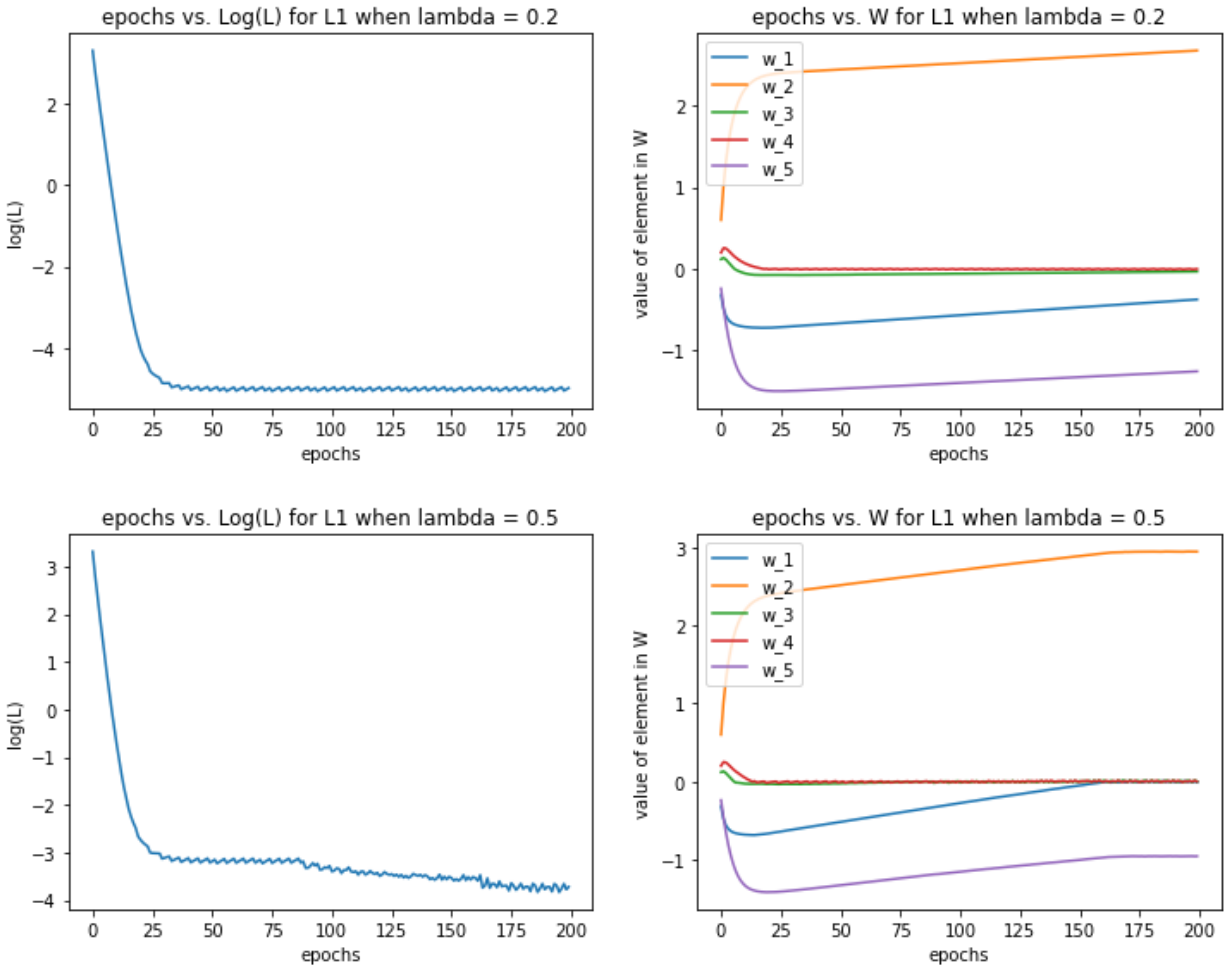
In [3]:
```
do_c(X,Y,W0,u,epochs)
```

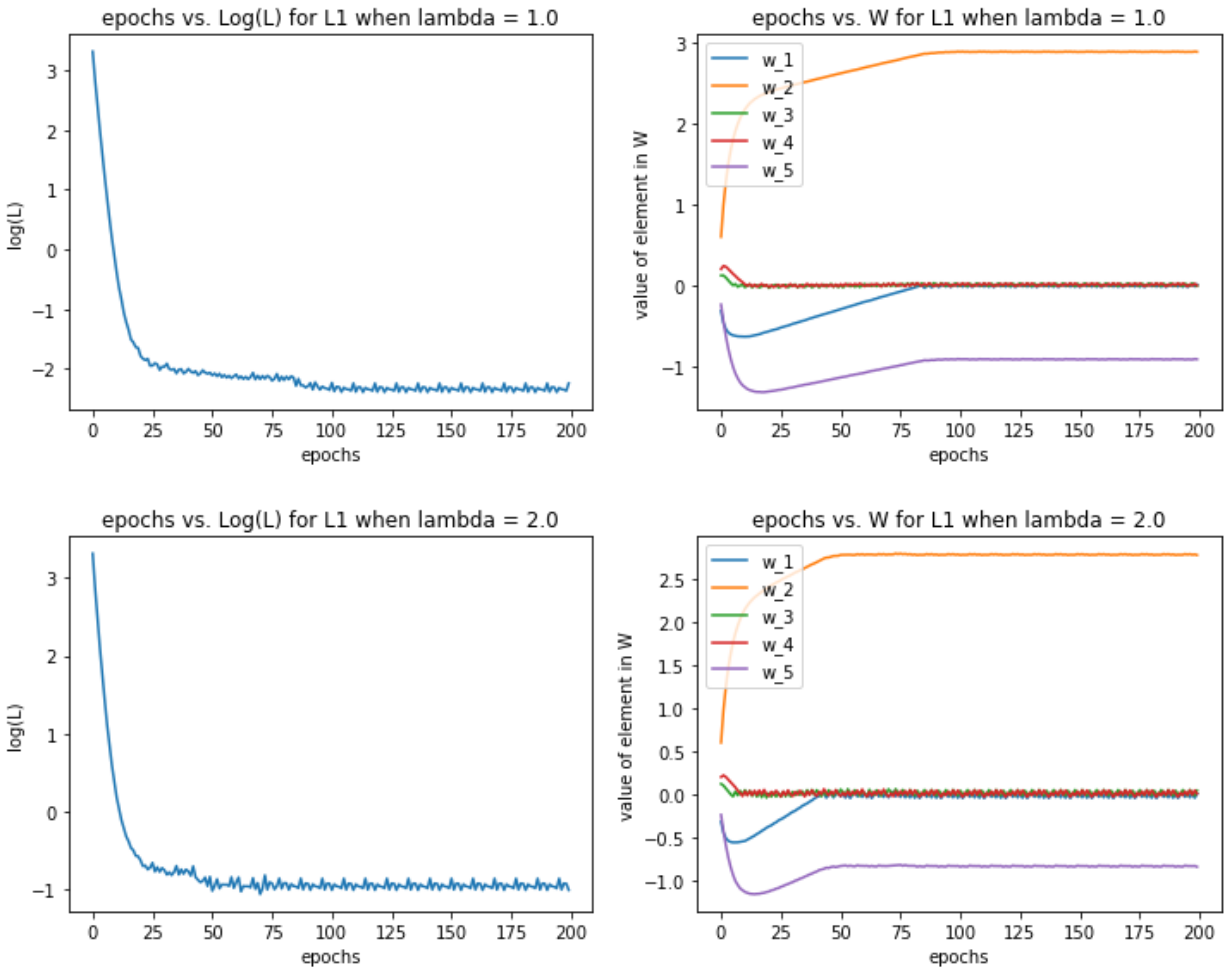


(d)

From the figure we can see that performace of L1 is not good, based on λs given by the question. For $\lambda = 0.2$, it converges to a point instead of an optimum and then oscillate around this point but we can see two weights of W are converging to 0. As we increase λ, we can see the we can have three weights are converging and oscillating at 0 but the loss function reach a point with larger loss which is not good. In conclusion, it is important for us to choose a good λ. If λ is too large, we will have a more sparse solution but not far from an optimum. If λ is too small, we may have a less sparse solution but closer to an optimum.

In [4]:
```
do_d(X,Y,W0,u,epochs)
```

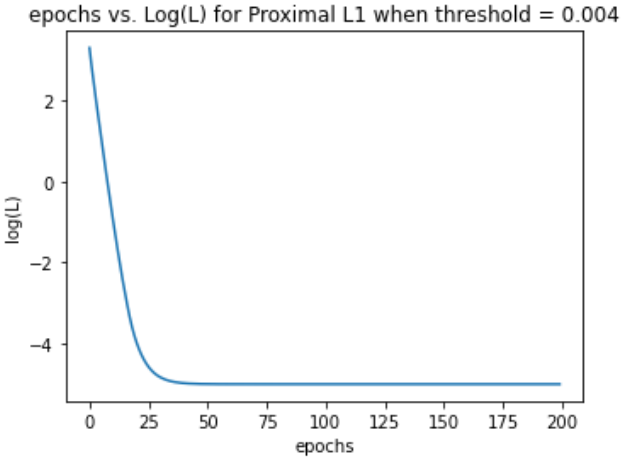


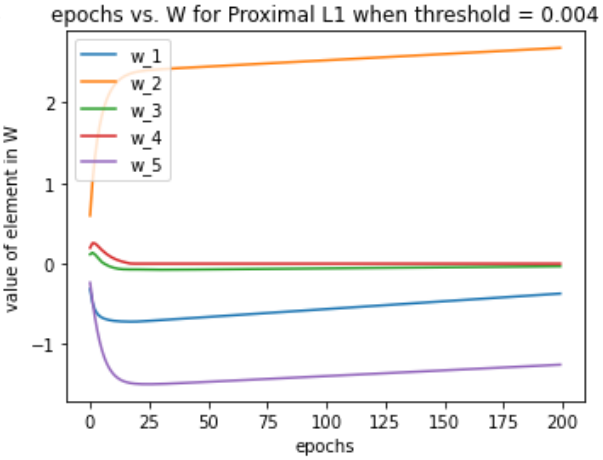


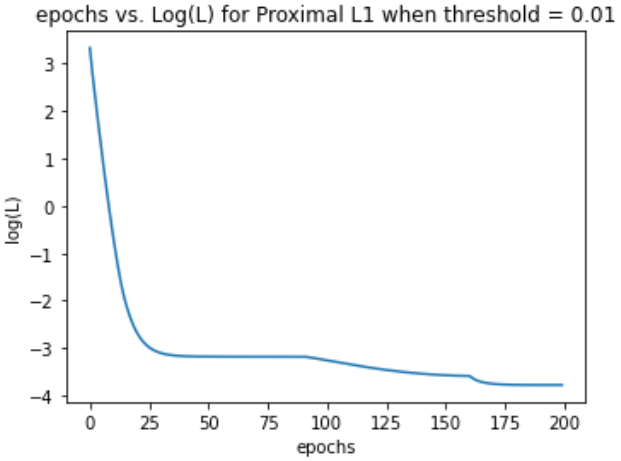
Lab1 (e)

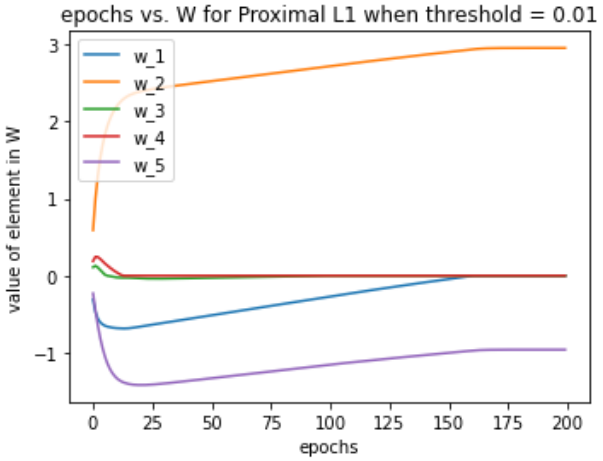
Compared to the results in (d), we can notice both lines of Loss and weigths are smoother. This is because the added proximity term will allow smoother convergence of the overall objective.

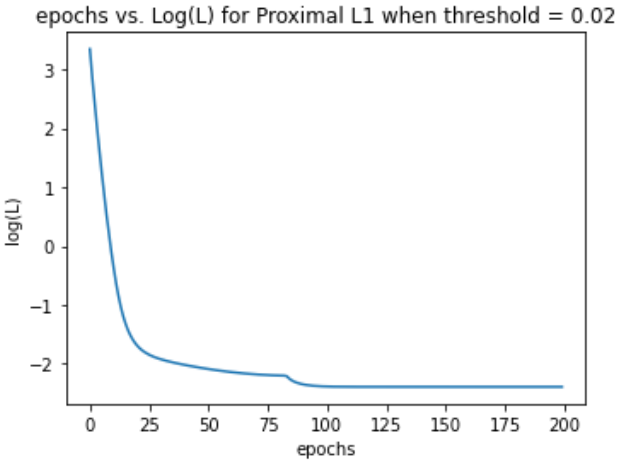
In [5]:
```
do_e(X,Y,W0,u,epochs)
```

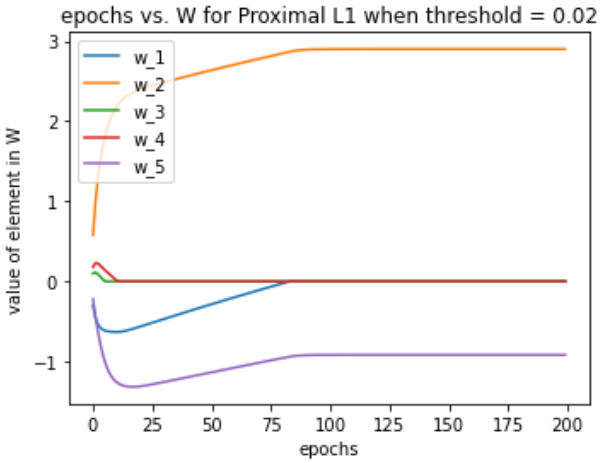

epochs vs. Log(L) for Proximal L1 when threshold = 0.004
log(L)
epochs

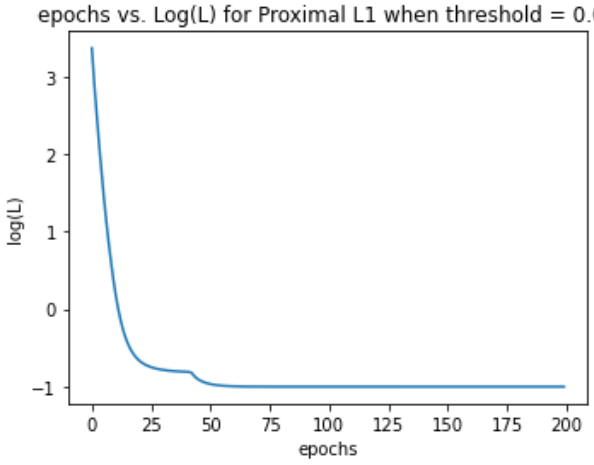

epochs vs. W for Proximal L1 when threshold = 0.004
w_1
w_2
w_3
w_4
w_5
value of element in W
epochs

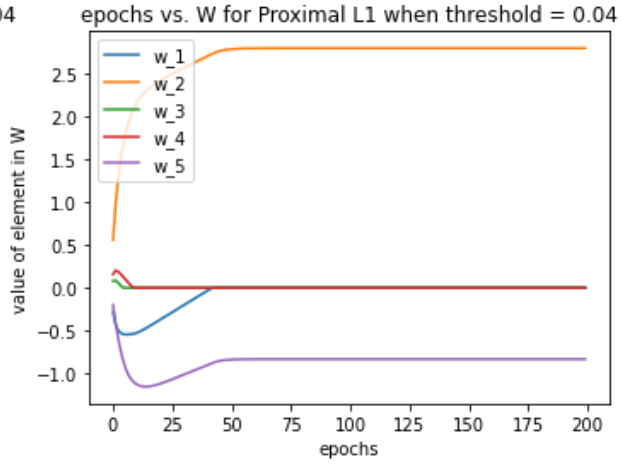

epochs vs. Log(L) for Proximal L1 when threshold = 0.01
log(L)
epochs


epochs vs. W for Proximal L1 when threshold = 0.01
w_1
w_2
w_3
w_4
w_5
value of element in W
epochs


epochs vs. Log(L) for Proximal L1 when threshold = 0.02
log(L)
epochs


epochs vs. W for Proximal L1 when threshold = 0.02
w_1
w_2
w_3
w_4
w_5
value of element in W
epochs



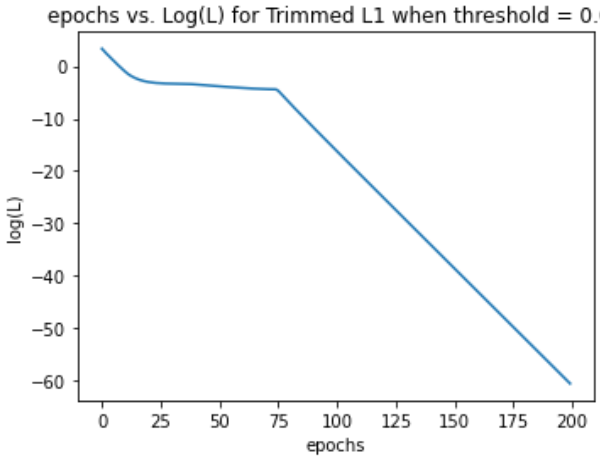


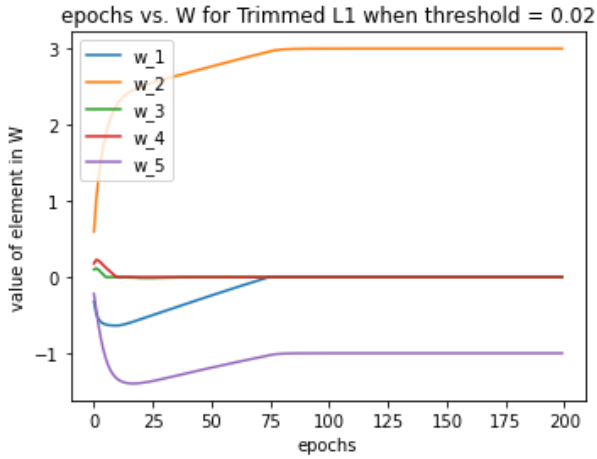
Lab1 (f)

(1) Comparison between trimmed l1 and l1: Trimmed l1 works much better than l1 on convergence. From figures in (d) and (f), we can see Trimmed l1 can converge to the optimum excpet for threshold = 0.02. But l1 cannot converge to an optimum for all λs. Besides, weights in trimmed l1 also converges much faster than l1. As we improve the threshold, we can see the speed of convergence for trimmed l1 also increase.

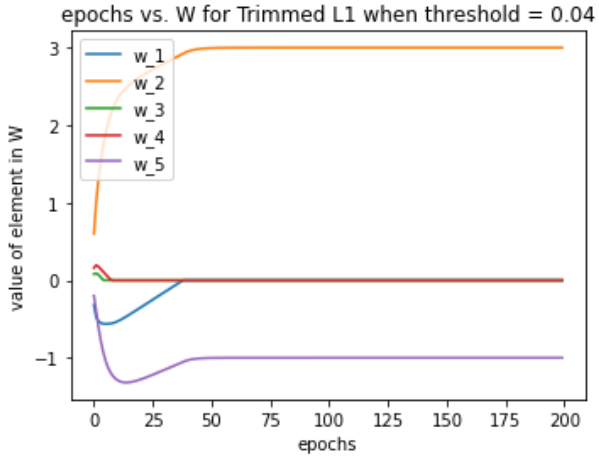
(2) Comparison between trimmed l1 and iterative prunning: From figures in (c) and (f),for the early steps (before 30 steps),when threshold = 0.02, 0.04, 0.1, the convergence speed of iterative pruning is faster than trimmed l1: At the step30, we can see log(loss) of iterative pruning is below -10 while trimmed l1 is above -10. But we can see that both trimmed l1 and iterative pruning can converge to an optimum with the same sparsity in the end except for threshold = 0.02.

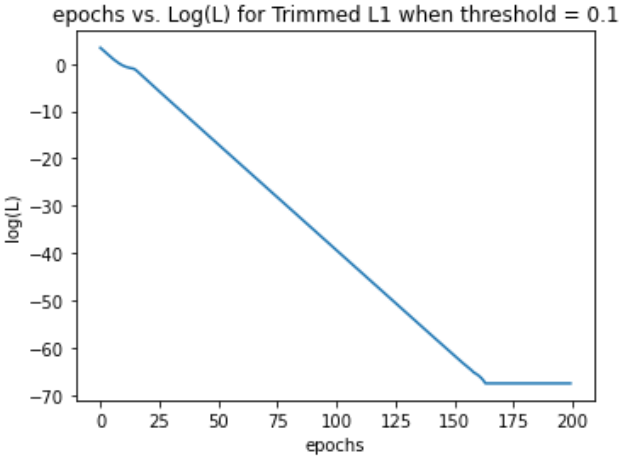
In [6]:
```
do_f(X,Y,W0,u,epochs)
```

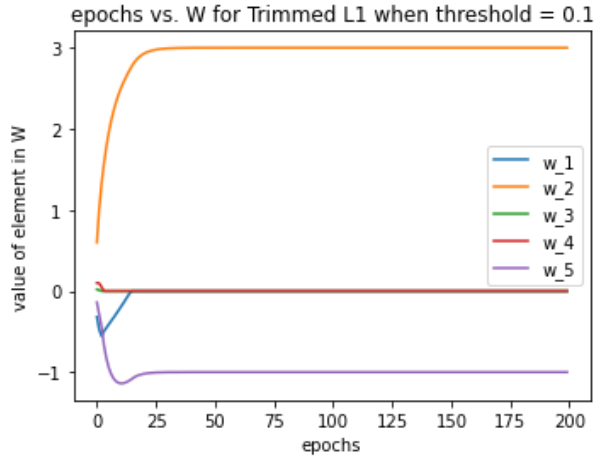


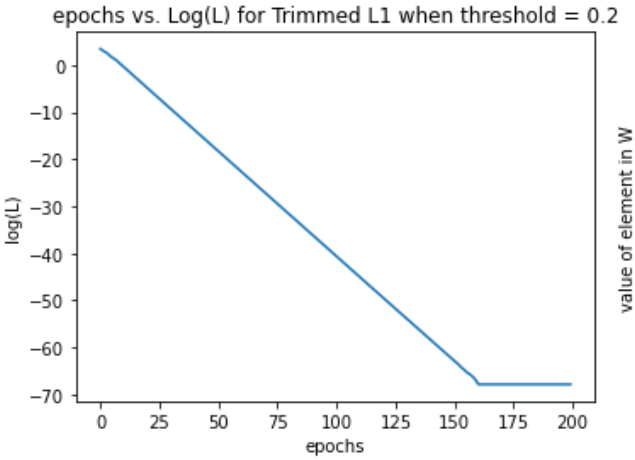


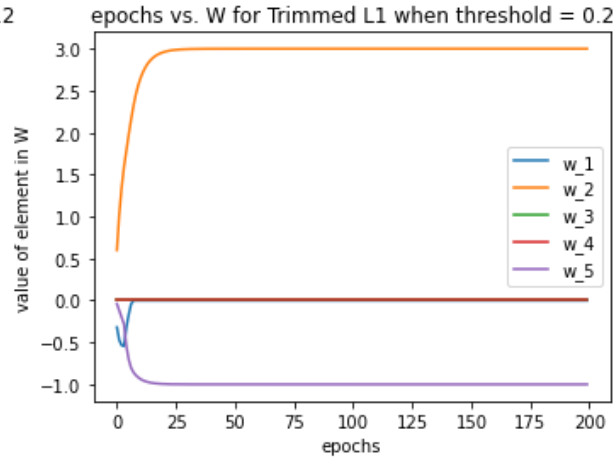

epochs vs. Log(L) for Trimmed L1 when threshold = 0.04
log(L)
epochs


epochs vs. W for Trimmed L1 when threshold = 0.04
value of element in W
epochs
w_1
w_2
w_3
w_4
w_5


epochs vs. Log(L) for Trimmed L1 when threshold = 0.1
log(L)
epochs


epochs vs. W for Trimmed L1 when threshold = 0.1
value of element in W
epochs
w_1
w_2
w_3
w_4
w_5


epochs vs. Log(L) for Trimmed L1 when threshold = 0.2
log(L)
epochs


epochs vs. W for Trimmed L1 when threshold = 0.2
value of element in W
epochs
w_1
w_2
w_3
w_4
w_5

Lab1 (g) Bonus

Analysis below are based on all figures in (c) -- (d), please go back to previous pages to see the figures in three data points. Here I only plot figures for two points in the following.

(c) For iterative pruning, when we use two data points, we have worse convergence performance than three data points. It seems that we reach a local minima and cannot jump out of this local minima.

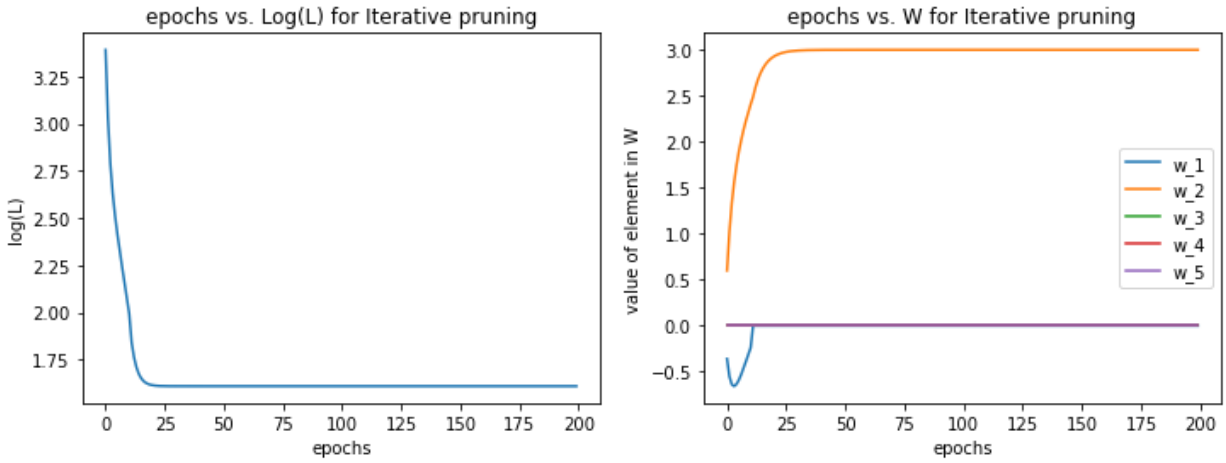
(d) For l1, when we use two data points, our loss objective are smoother than that of three data points when threshold smaller than 0.5 but more osciallted than that of three data points when threshold larger than 0.5. For weights objective, three data points have a better performance than that of two data points. We can see that for weights objective, three weights can always converge or oscillate around 0 while it does not work for two data points.

(e) For prixmal l1, when we use two data points, it has worse performace on sparsity of weights than that of three data points where we can always have 3 zero weights in the end.

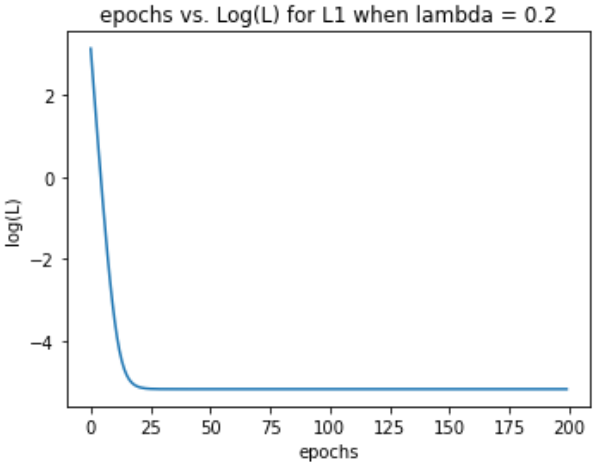
(f) For trimmed l1, three data points have a better performace on convergence. We can notice that both of the loss and weights objectives converges faster much than that of two points.

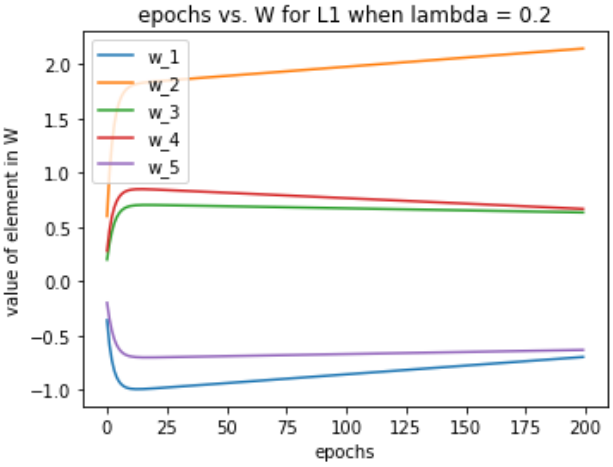
In [7]:
```
X_new =np.array([[1,-2,-1,-1,1],[2, -1, 2, 0, -2]])
Y_new = np.array([-7,-1])
W0 = np.array([0,0,0,0,0]).T
u = 0.02
epochs = 200
```

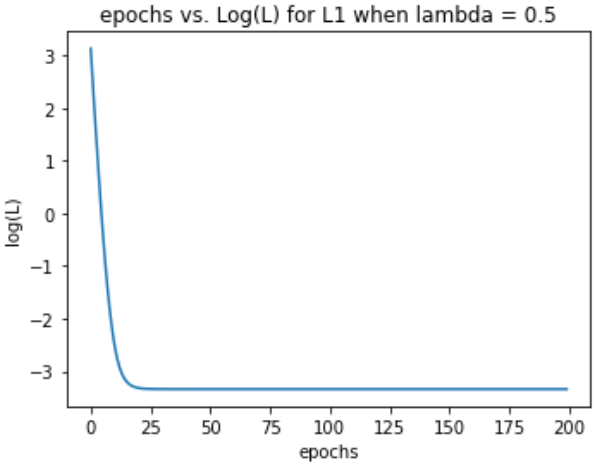
In [8]:
```
do_c(X_new,Y_new,W0,u,epochs)
```

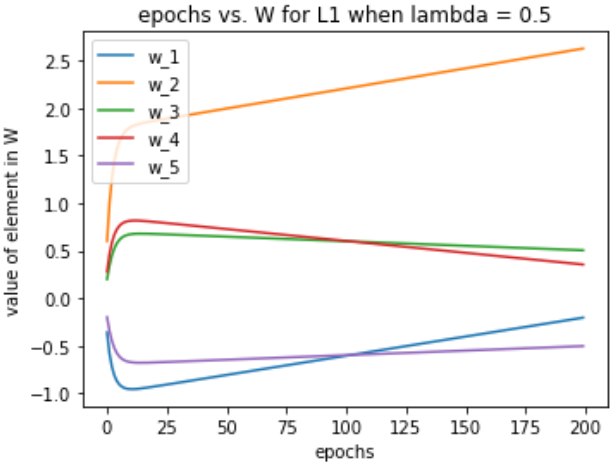


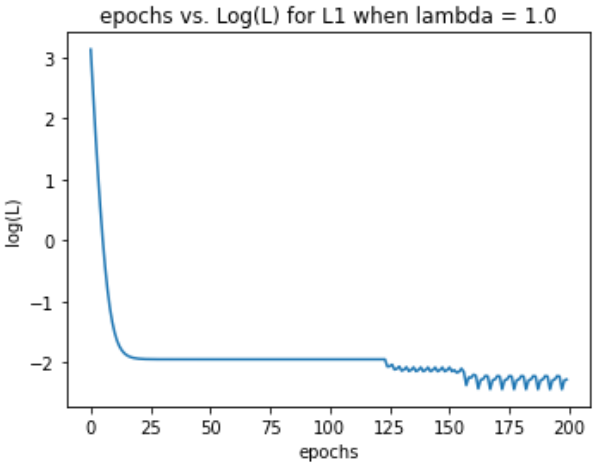
In [9]:
```
do_d(X_new,Y_new,W0,u,epochs)
```

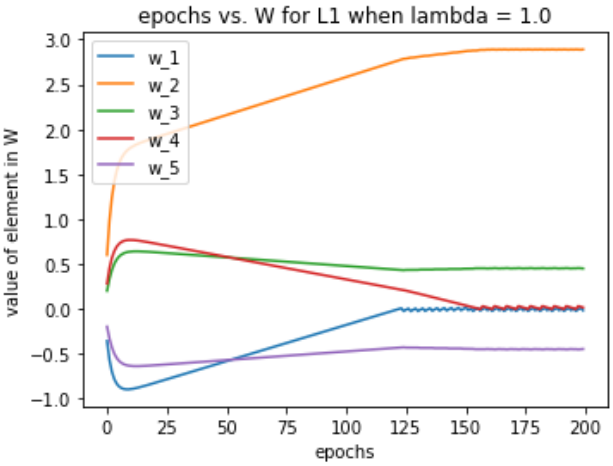

epochs vs. Log(L) for L1 when lambda = 0.2
log(L)
epochs

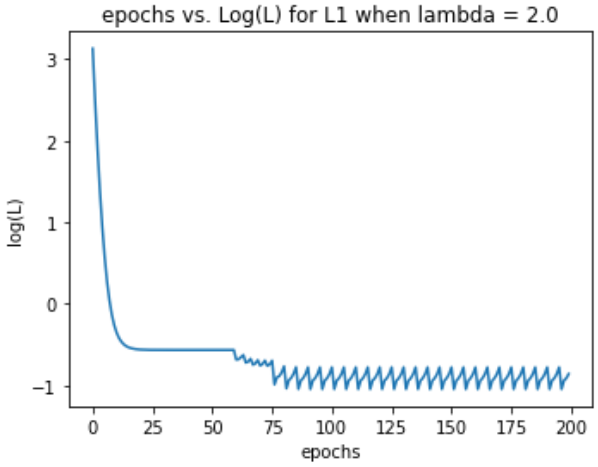

epochs vs. W for L1 when lambda = 0.2
w_1
w_2
w_3
w_4
w_5
value of element in W
epochs

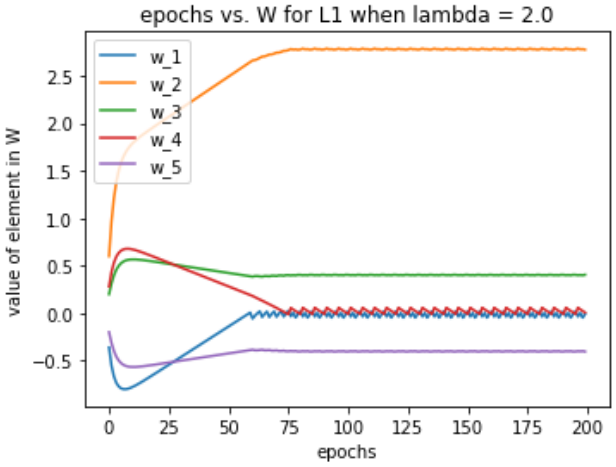

epochs vs. Log(L) for L1 when lambda = 0.5
log(L)
epochs


epochs vs. W for L1 when lambda = 0.5
w_1
w_2
w_3
w_4
w_5
value of element in W
epochs

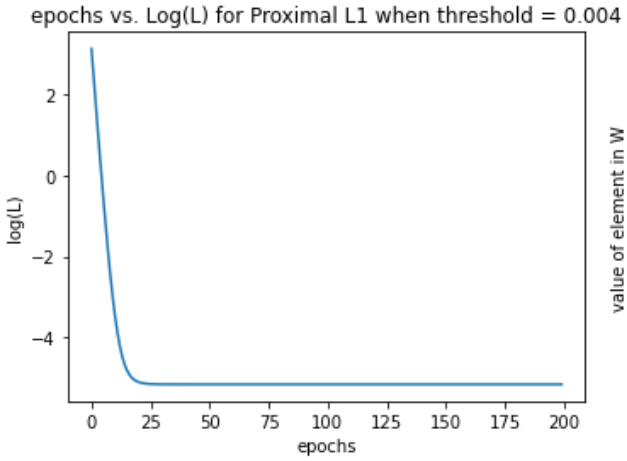

epochs vs. Log(L) for L1 when lambda = 1.0
log(L)
epochs

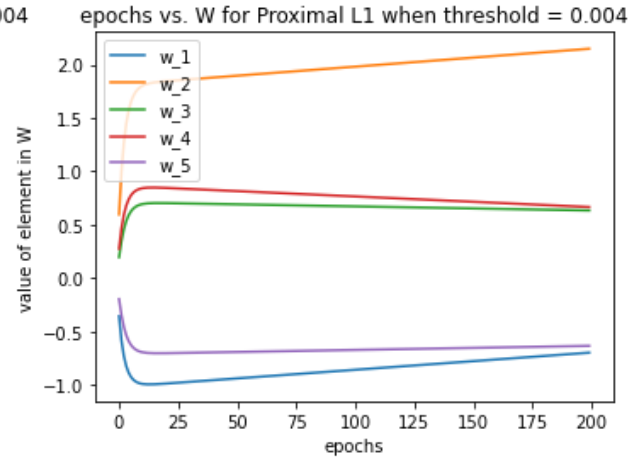

epochs vs. W for L1 when lambda = 1.0
w_1
w_2
w_3
w_4
w_5
value of element in W
epochs

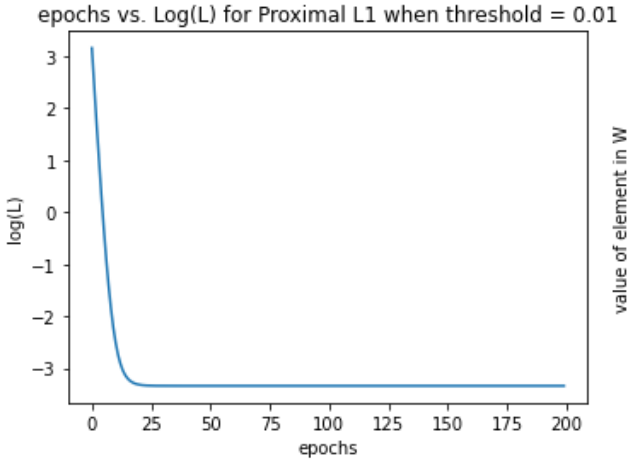


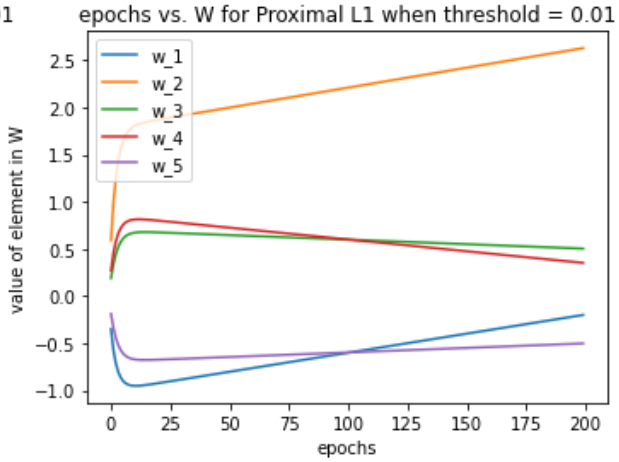


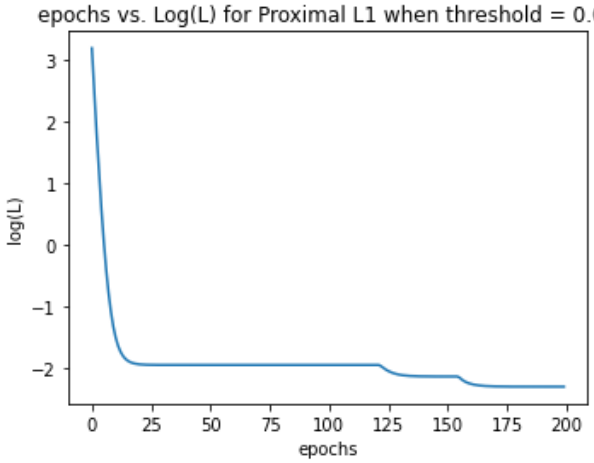
In [10]:
```
do_e(X_new,Y_new,W0,u,epochs)
```

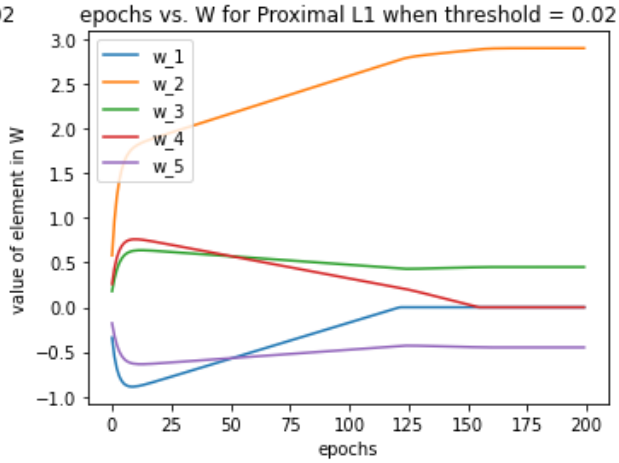


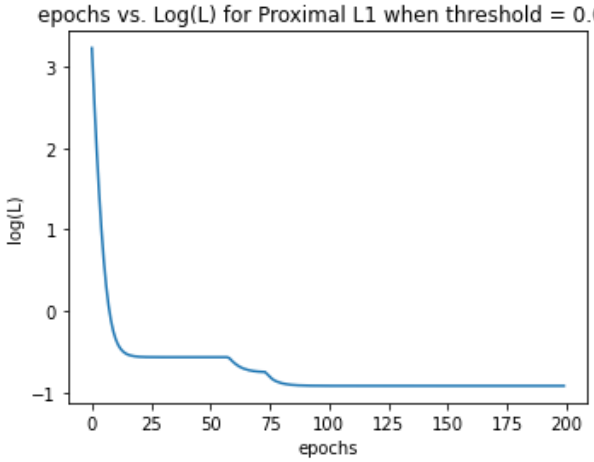


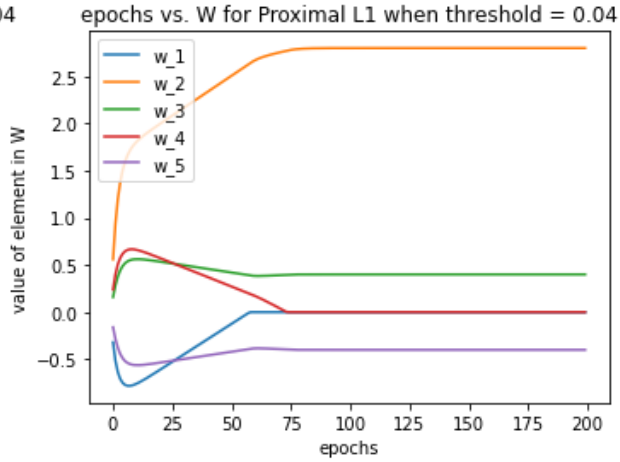




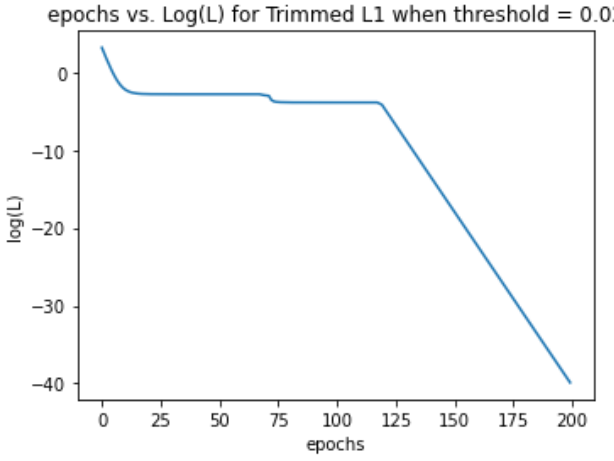


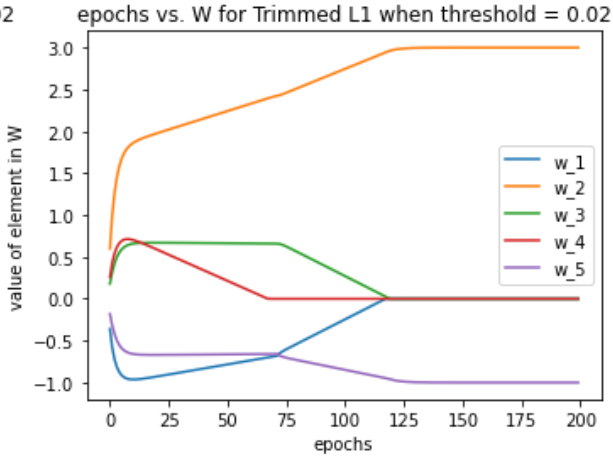


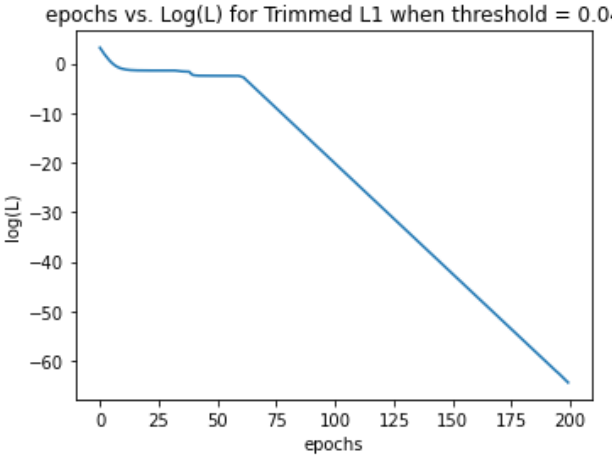


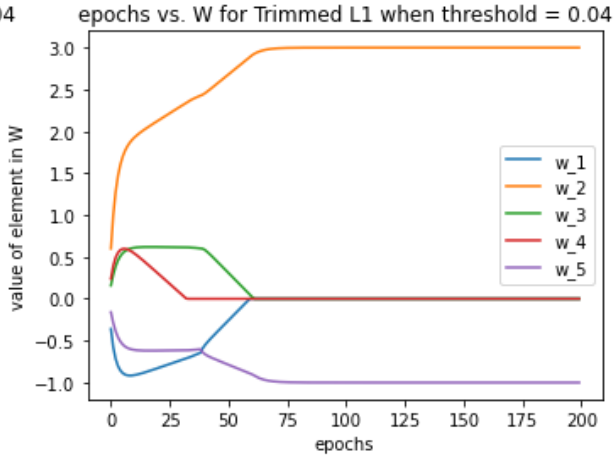


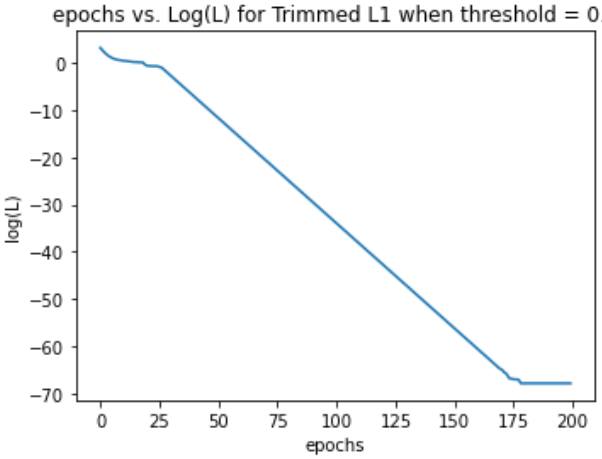
In [11]:
```
do_f(X_new,Y_new,W0,u,epochs)
```

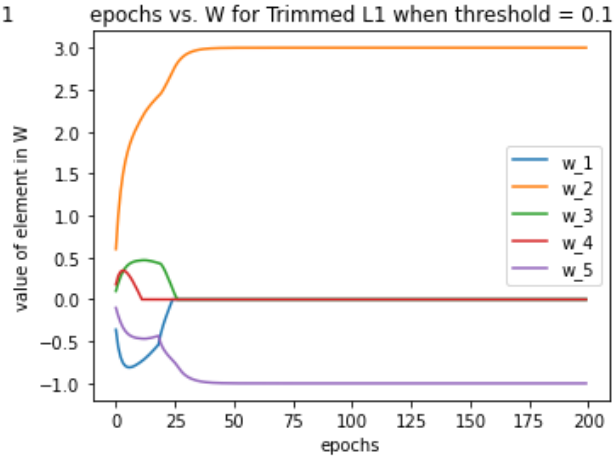


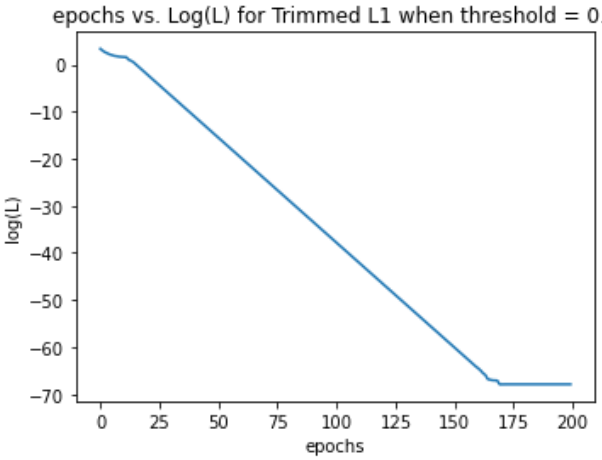


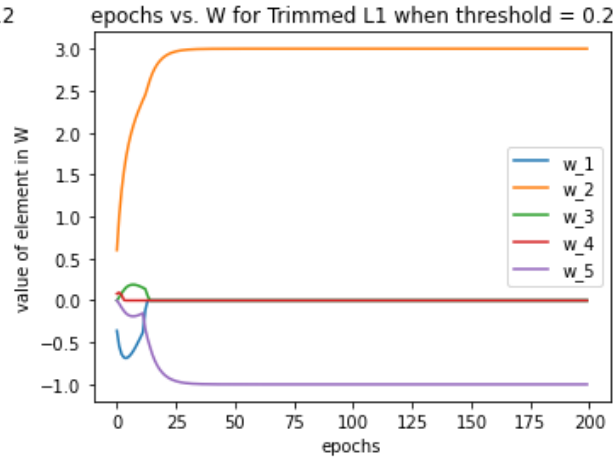

epochs vs. Log(L) for Trimmed L1 when threshold = 0.04
log(L)
epochs


epochs vs. W for Trimmed L1 when threshold = 0.04
value of element in W
epochs
w_1
w_2
w_3
w_4
w_5


epochs vs. Log(L) for Trimmed L1 when threshold = 0.1
log(L)
epochs


epochs vs. W for Trimmed L1 when threshold = 0.1
value of element in W
epochs
w_1
w_2
w_3
w_4
w_5


epochs vs. Log(L) for Trimmed L1 when threshold = 0.2
log(L)
epochs


epochs vs. W for Trimmed L1 when threshold = 0.2
value of element in W
epochs
w_1
w_2
w_3
w_4
w_5

Lab2

Lab2 (a)

```
class STE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, w, bit):  # "w" is the weight tensor (as a PyTorch Tensor), "bit" is the precision to be quantized to. You can ign
"ctx" as it will not be used
        if bit is None:
            wq = w         # Full precision, no change
        elif bit==0:
            wq = w*0       # Zero precision, everything is zero
        else:
            # Your code here:
            alpha =  torch.max(w)  - torch.min(w)    # Compute alpha (scale) for dynamic scaling
            beta =   torch.min(w)      # Compute beta (bias) for dynamic scaling
            ws =  (w - beta)/alpha   # Scale w with alpha and beta so that all elements in ws are between 0 and 1
            R =   torch.round((2**bit - 1)* ws)/(2**bit - 1)      # Quantize ws with a linear quantizer to "bit" bits
            wq =   R*alpha + beta      # Scale the quantized weight R back with alpha and beta
            # End of your code, do not change below this line
        return wq

    @staticmethod
    def backward(ctx, g):
        return g, None
```

Lab2 (b

Bits	Test Accuracy
None	0.9151
6	0.9145
5	0.9112
4	0.8972
3	0.7662
2	0.0899

Lab2 (c)

We can see as the bits is decreasing, which means precision is decreasing, the test accuracy is decreasing. And from the expriments result (c) and (b), we can notice that finetune is very effective abd imporve the accuracy a lot.

Bits	Highest Test Accuracy
4	0.9130
3	0.9045
2	0.8650

Lab2 (d)

Bits	Test Accuracy Before finetune	Test Accuracy After finetune
{4,4,2}	0.4727	0.8907
{4,2,4}	0.4157	0.8935

{2,4,4}	0.1041	0.9052

Lab2 (e)

[30]:
```
num_1 = 3*16*3*3   # number of weights in conv1
num_s1 = 16*16*3*3*6  # the number of weights in stage1
num_s2 = 16*32*3*3 + 32*32*3*3*5 + 16*32*1*1  # the number of weights in stage2
num_s3 = 32*64*3*3 + 64*64*3*3*5 + 32*64*1*1 # the number of weights in stage3
num_20 = 64*10 # the number of weights in final layer

sum_num = num_1 + num_s1+ num_s2+ num_s3+ num_20  # total number of weights for all layers

bits_442 = num_1*32 + num_s1*4+ num_s2*4+num_s3*2+num_20*32 # total bits of all layers for set {442}
print("{4, 4, 2} precision:" + str(bits_442/sum_num))
bits_424 = num_1*32 + num_s1*4+ num_s2*2+num_s3*4+num_20*32 # total bits of all layers for set {424}
print("{4, 2, 4} precision:" + str(bits_424/sum_num))
bits_244 = num_1*32 + num_s1*2+ num_s2*4+num_s3*4+num_20*32 # total bits of all layers for set {244}
print("{2, 4, 4} precision:" + str(bits_244/sum_num))
```

```
{4, 4, 2} precision:2.598783296910992
{4, 2, 4} precision:3.7327978264721517
{2, 4, 4} precision:4.008741361998701
```

Lab2 (f)

Combine the results in (d) and (e), I think the scheme {4,4,2} leads to the best accuracy- average precision tradeoff. We can notice that three schemes have similar test accuracy (no ver big gap). But average precision of {4,4,2} is only 2.6 bits, which can reduce a lot cost. The first stage is the least important for reaching high accuracy. We can see from (f) and (e), although we have 2 bits to the first stage, we can still have high accuracy after finetuning.

In [ ]: