

▼ Brief Summary

In this paper, I implemented a feed forward neural network using only numpy. Different activation functions have their own classes, where the forward and backward method implements the feedforward and backpropagation step. Similarly as an output of the final layer, we have separate class for Softmax operation. Another class is designed for maintaining the linear hidden layers.

In the final model class, we can choose the type and structure of the hidden layers, activation functions between the hidden layers and the final output for the final layers. In addition we also need to provide the cost function which will be used to generate loss for backpropagation.

I used MNIST dataset to classify the digit image. This is a classification problem, so we used softmax function at the final layer which will produce a probability for each of the output layer neuron. For primary investigation, we used a network of 2 hidden layers with 128 and 64 neurons. Just by changing the layers attribute of the model class, we can easily modify our network structures. We used Leaky ReLU as activation function as they do not suffer from the vanishing gradient (like sigmoid) or dead neuron (like ReLU) problem. Similarly to avoid vanishing gradient problem, we used He weight initialization. Multi-class cross entropy is used as the loss function. Our model is capable to use the L2 regularization and dropout. Similarly it is designed to incorporate SGD, Nesterov Momentum, RMSProp and Adam optimizer.

Backpropagation starts from the cost function, then it follows backward to the Softmax, hidden layers and activation function. So other than the cost function, every backward method takes the input of the gradient wrt the next layer and outputs the gradient wrt the previous layer.

In primary investigation with only SGD and a learning rate of 0.5, the model completed 10 epoch in 5 min with a validation loss of 0.13.

▼ Export Data

```
1 from sklearn.datasets import fetch_openml
2 import numpy as np
3 from sklearn.model_selection import train_test_split
4 import time
5 import matplotlib.pyplot as plt
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```
2 y = np.array(y_orig, dtype = 'int')
3 targets = np.eye(10)[y]
```

▼ Activation Functions

```
1 class Sigmoid():
2     def forward(self, x):
3         self.sigmoid = 1. / (1 + np.exp(-x))
4         return self.sigmoid
5
6     def backward(self, gradient):
7         return self.sigmoid * (1 - self.sigmoid) * gradient
8
9 class ReLU():
10    def forward(self, x):
11        self.x = x
12        return (x>0)*1
13
14    def backward(self, gradient):
15        return (self.x > 0) * gradient
16
17 class LeakyReLU():
18    def __init__(self, alpha = 0.01):
19        self.alpha = alpha
20
21    def forward(self, x):
22        self.x = x
23        return (x>0)*x + (x<0)*self.alpha*x
24
25    def backward(self, gradient):
26        return ( (self.x>0) + (self.x<0)*self.alpha ) * gradient
```

Double-click (or enter) to edit

▼ Output Layer

```
1 class Softmax():
```

```

2     def forward(self, x):
3         exps = np.exp(x - x.max())
4         self.output = exps / np.sum(exps, axis = 1)[: ,None]
5         return self.output
6
7     def backward(self, gradient):
8         return self.output * (gradient -
9                               (gradient * self.output).sum(axis=1)[: ,None] )
10

```

▼ Loss Function

```

1 class CrossEntropy():
2     def forward(self, y_pred, y_true):
3         self.y_pred = y_pred
4         self.y_true = y_true
5         return np.sum(-1. * y_true * np.log(y_pred)) / y_pred.shape[0]
6
7     def backward(self):
8         return -1. * self.y_true / self.y_pred

```

▼ Linear Layers

```

1 (np.random.rand(5) < 0.5) * np.array([1,2,3,4,5])

array([1, 2, 0, 4, 0])

```

```

1 class Linear():
2     def __init__(self, n_input, n_output, dropout_p = 0):
3         # He weight initialization as default activation is ReLU
4         self.Ws = np.random.randn(n_input, n_output) * np.sqrt(2/n_input)
5         self.bs = np.zeros(n_output)
6         # parameters for momentum optimizer
7         self.vw = np.zeros((n_input, n_output))
8
9         self.vb = np.zeros(n_output)
10        # parameters for RMSprop optimizer
11        self.grad_sq_w = np.zeros((n_input, n_output))
12        self.grad_sq_b = np.zeros(n_output)

```

```

11     self.grad_sq_w = np.zeros(n_output)
12     # parameters for adam optimizer
13     self.moment_1_w = np.zeros((n_input, n_output))
14     self.moment_2_w = np.zeros((n_input, n_output))
15     self.moment_1_b = np.zeros(n_output)
16     self.moment_2_b = np.zeros(n_output)
17     self.dropout_p = dropout_p # dropout probability
18
19     def forward(self, x):
20         self.x = x
21         # dropout layer: 0 means it will be dropped away
22         dropout_layer = np.random.rand(len(self.bs)) > self.dropout_p
23         return (np.dot(x, self.Ws) + self.bs) * dropout_layer
24
25     def backward(self, gradient):
26         self.grad_b = gradient.mean(axis=0)
27         self.grad_W = (self.x[:, :, None] @ gradient[:, None, :]).mean(axis=0)
28         return np.dot(gradient, self.Ws.transpose())

```

▼ Full Model

```

1 class Model():
2     def __init__(self, layers, cost_func, n_epoch=10, lr=0.01,
3                 reg_lambda=0, beta = 0, decay_rate = 0,
4                 beta1 = 0, beta2 = 0, dropout_p = 0):
5         self.layers = layers
6         self.cost = cost_func
7         self.n_epoch = n_epoch
8         self.lr = lr # learning rate
9         self.beta = beta # momentum parameter
10        self.reg_lambda = reg_lambda # regularization parameter
11        self.decay_rate = decay_rate # rmsprop decay parameter
12        # Adam moment parameter
13        self.beta1 = beta1
14        self.beta2 = beta2
15        self.dropout_p = dropout_p # dropout probability
16
17    def forward(self, x):
18        for layer in self.layers:
19            if type(layer) == Linear:
20                layer.dropout_p = self.dropout_p
21                # layer.forward(x)

```

```

21         x = layer.forward(x)
22     return x
23
24     def loss(self, input, y_true):
25         y_pred = self.forward(input)
26         return self.cost.forward(y_pred, y_true)
27
28     def backward(self):
29         gradient = self.cost.backward()
30         n_layer = len(self.layers)
31         for i in range(n_layer-1, -1, -1):
32             gradient = self.layers[i].backward(gradient)
33
34     def make_minibatch(self, x_data, y_data, mb_size):
35         minibatch_data = [(x_data[i:i+mb_size], y_data[i:i+mb_size])] \
36             for i in range(0, len(x_data), mb_size)]
37         return minibatch_data
38
39     def train(self, features, targets, mb_size, test_size = 0.20,
40               optimizer = 'nesterov'):
41         x_train, x_val, y_train, y_val = train_test_split(features, targets, test_size=test_size,
42                                                             random_state=42)
43         minibatch_data = self.make_minibatch(x_train, y_train, mb_size)
44         self.train_loss = []
45         self.test_loss = []
46         for epoch in range(self.n_epoch):
47             current_loss = 0
48             n_batch = len(minibatch_data)
49             for minibatch in minibatch_data:
50                 inputs, labels = minibatch
51                 current_loss += self.loss(inputs, labels)
52                 self.backward()
53                 # paramter updates
54                 for layer in self.layers:
55                     if type(layer) == Linear:
56                         if optimizer == 'nesterov':
57                             self.nesterov_momentum(layer)
58                         elif optimizer == 'rmsprop':
59                             self.rmsprop(layer)
60                         elif optimizer == 'adam':
61                             self.adam(layer, epoch)
62             self.train_loss.append(current_loss/n_batch)
63             self.test_loss.append(self.loss(x_val, y_val))
64             if (epoch+1) % 5 == 0:

```

```

64         if (epoch+1) % 5 == 0:
65             print(f'Epoch {epoch+1}/{self.n_epoch}: loss = {self.train_loss[epoch]}')
66
67     def loss_curve(self):
68         plt.plot(range(1, self.n_epoch+1), self.train_loss, label = 'train')
69         plt.plot(range(1, self.n_epoch+1), self.test_loss, label = 'test')
70         plt.legend()
71         plt.xlabel('Epoch')
72         plt.ylabel('Cross Entropy Loss')
73         plt.title('Loss Curve')
74
75     def nesterov_momentum(self, layer):
76         '''
77         for beta = 0, it becomes a normal SGD
78         for reg_alpha = 0, there will be no regularization
79         '''
80         vw_old = layer.vw
81         # velocity update with regularization (weight decay)
82         layer.vw = self.beta * vw_old - self.lr * layer.grad_W \
83             - self.lr * self.reg_lambda * layer.Ws
84         # position Update
85         layer.Ws += (1+self.beta) * layer.vw - self.beta * vw_old
86
87         vb_old = layer.vb
88         # velocity update
89         layer.vb = self.beta * vb_old - self.lr * layer.grad_b
90         # position Update
91         layer.bs += (1+self.beta) * layer.vb - self.beta * vb_old
92
93     def rmsprop(self, layer):
94         layer.grad_sq_w = self.decay_rate * layer.grad_sq_w \
95             + (1 - self.decay_rate) * layer.grad_W**2
96         layer.Ws += - self.lr * layer.grad_W / (np.sqrt(layer.grad_sq_w) + 1e-7)
97
98         layer.grad_sq_b = self.decay_rate * layer.grad_sq_b \
99             + (1 - self.decay_rate) * layer.grad_b**2
100         layer.bs -= self.lr * layer.grad_b / (np.sqrt(layer.grad_sq_b) + 1e-7)
101
102     def adam(self, layer, epoch):
103         # moment update
104         layer.moment_1_w = self.beta1 * layer.moment_1_w \
105             + (1 - self.beta1) * layer.grad_W
106         layer.moment_2_w = self.beta2 * layer.moment_2_w \
107             + (1 - self.beta2) * layer.grad_W**2

```

```

107         + (1 - self.beta2) * layer.grad_w**2
108     # bias correction
109     unbiased_moment_1_w = layer.moment_1_w / (1 - self.beta1 ** (epoch+1))
110     unbiased_moment_2_w = layer.moment_2_w / (1 - self.beta2 ** (epoch+1))
111     # update parameter
112     layer.ws -= self.lr * unbiased_moment_1_w / (np.sqrt(unbiased_moment_2_w) + 1e-7)
113
114     # moment update
115     layer.moment_1_b = self.beta1 * layer.moment_1_b \
116         + (1 - self.beta1) * layer.grad_b
117     layer.moment_2_b = self.beta2 * layer.moment_2_b \
118         + (1 - self.beta2) * layer.grad_b**2
119     # bias correction
120     unbiased_moment_1_b = layer.moment_1_b / (1 - self.beta1 ** (epoch+1))
121     unbiased_moment_2_b = layer.moment_2_b / (1 - self.beta2 ** (epoch+1))
122     # update parameter
123     layer.bs -= self.lr * unbiased_moment_1_b / (np.sqrt(unbiased_moment_2_b) + 1e-7)
124

```

▼ Testing Working Model

▼ SGD

```

1 model = Model([Linear(784,128), LeakyReLU(), Linear(128,64), LeakyReLU(), Linear(64,10), Softmax()], CrossEntropy())
2
3 model.n_epoch = 1
4 model.lr = 0.1
5 model.reg_lambda = 0
6
7 mb_size = 50
8 model.train(inputs, targets, mb_size,
9             optimizer='nesterov')

```

```

1 model.n_epoch = 1
2 model.test_loss[-1], model.train_loss[-1]

```

```

(0.18076048213141493, 0.3187320765505537)

```

```

1 # we will use test dataset for the unbiased evaluation of the final model

```

```

1 # random search 1-fold cross_validation
2 x_train, x_test, y_train, y_test = train_test_split(inputs, targets, test_size
3                                                    random_state=42)

```

```

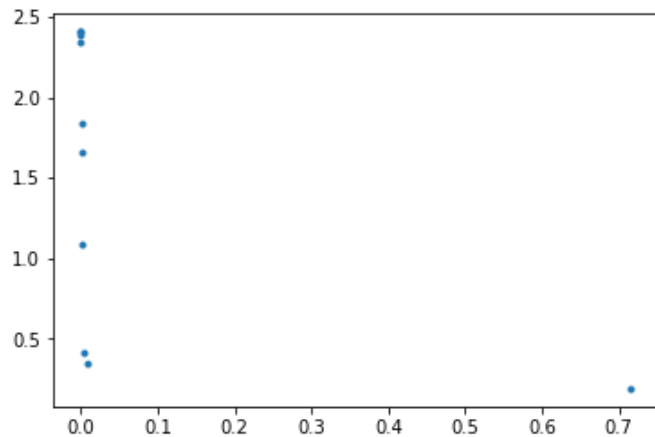
1 # random search 1-fold cross_validation
2 n_iterations = 10
3 validation_loss = []
4 lr_s = []
5 for _ in range(n_iterations):
6     model = Model([Linear(784,128), LeakyReLU(), Linear(128,64), LeakyReLU(),
7                     Linear(64,10), Softmax()], CrossEntropy())
8     model.lr = 10*np.random.uniform(-6,1)
9     model.n_epoch = 2
10    mb_size = 50
11    model.train(x_train, y_train, mb_size,
12               optimizer='nesterov')
13    validation_loss.append(model.test_loss[-1])
14    lr_s.append(model.lr)
15

```

```
1 plt.plot(lr_s, validation_loss, '.')

```

[<matplotlib.lines.Line2D at 0x7f464def8510>]



```

1 model = Model([Linear(784,128), LeakyReLU(), Linear(128,64), LeakyReLU(),
2               Linear(64,10), Softmax()], CrossEntropy())
3 model.lr = 0.5
4 model.n_epoch = 10
5 mb_size = 50

```

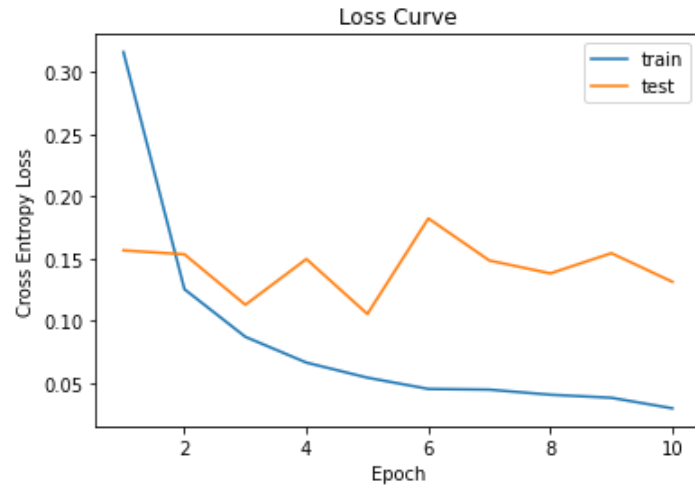


```

5 mb_size = 50
6 model.train(x_train, y_train, mb_size,
7             optimizer='nesterov')
8 model.loss_curve()

```

Epoch 5/10: loss = 0.05425715980876315
Epoch 10/10: loss = 0.02961947304835689



```

1 model.train_loss[-1], model.test_loss[-1]

(0.02961947304835689, 0.13130076481722563)

```

▼ SGD + Momentum

```

1 model = Model([Linear(784,128), ReLU(), Linear(128,64), ReLU(), Linear(64,10), Softmax()], CrossEntropy())
2
3 model.n_epoch = 50
4 model.lr = 0.2
5 model.reg_lambda = 0
6 model.beta = 0.9
7
8 mb_size = 50
9 model.train(inputs[:1000], targets[:1000], mb_size,
10             optimizer='nesterov')

```

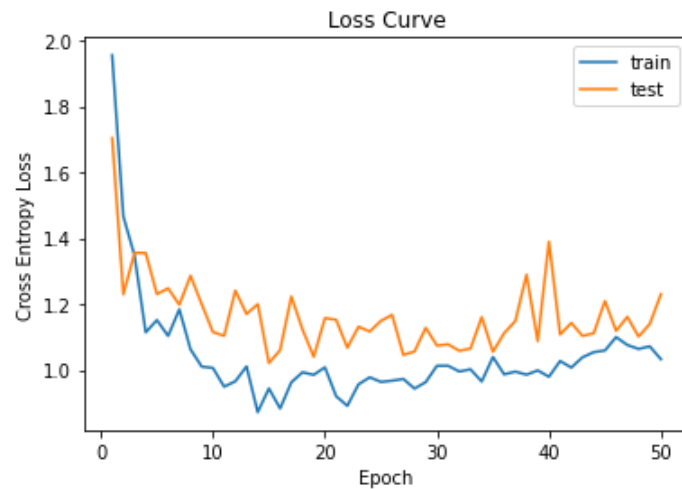
Epoch 5/50: loss = 1.1517014800007200

```

Epoch 5/50: loss = 1.1517014070007200
Epoch 10/50: loss = 1.00662743040652
Epoch 15/50: loss = 0.9443916198321197
Epoch 20/50: loss = 1.0084874563668824
Epoch 25/50: loss = 0.9643326139300877
Epoch 30/50: loss = 1.0130861220831628
Epoch 35/50: loss = 1.0398071627464067
Epoch 40/50: loss = 0.980003936129946
Epoch 45/50: loss = 1.0596051365987094
Epoch 50/50: loss = 1.0330920507013976

```

```
1 model.loss_curve()
```



▼ RMSprop

```

1 model = Model([Linear(784,128), ReLU(), Linear(128,64), ReLU(), Linear(64,10), Softmax()], CrossEntropy())
2
3 model.n_epoch = 50
4 model.lr = 0.1
5
6 model.decay_rate = 0.99
7 mb_size = 50

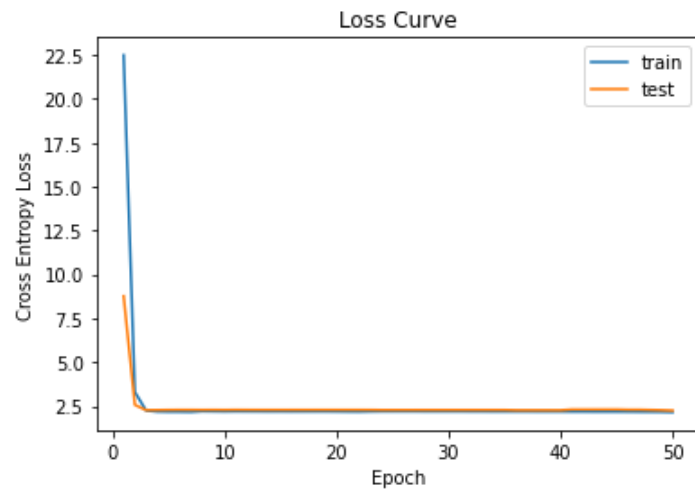
8 model.train(inputs[:1000], targets[:1000], mb_size,
9             optimizer='rmsprop')

Epoch 5/50: loss = 2.182070555078885

```

```
Epoch 5/50: loss = 2.16207555020003
Epoch 10/50: loss = 2.1988231122133968
Epoch 15/50: loss = 2.1985008803422024
Epoch 20/50: loss = 2.194965930453911
Epoch 25/50: loss = 2.2040840284984777
Epoch 30/50: loss = 2.201275018452942
Epoch 35/50: loss = 2.1953194477842866
Epoch 40/50: loss = 2.1929040052329642
Epoch 45/50: loss = 2.186529289103248
Epoch 50/50: loss = 2.161922242154096
```

```
1 model.loss_curve()
```

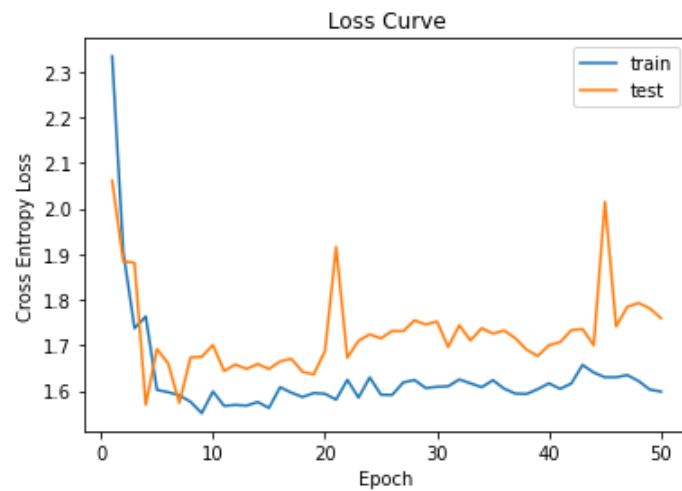


▼ Adam

```
1 model = Model([Linear(784,128), ReLU(), Linear(128,64), ReLU(), Linear(64,10), Softmax()], CrossEntropy())
2
3 model.n_epoch = 50
4 model.lr = 0.1
5
6 model.beta1 = 0.9
7 model.beta2 = 0.999
8
9 mb_size = 50
10 model.train(inputs[:1000], targets[:1000], mb_size,
11             optimizer='adam')
```

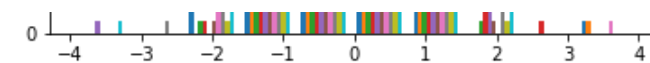
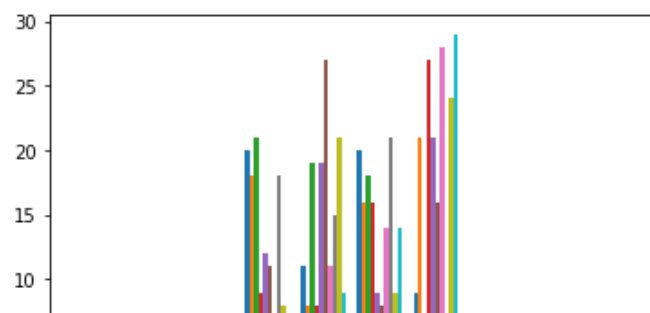
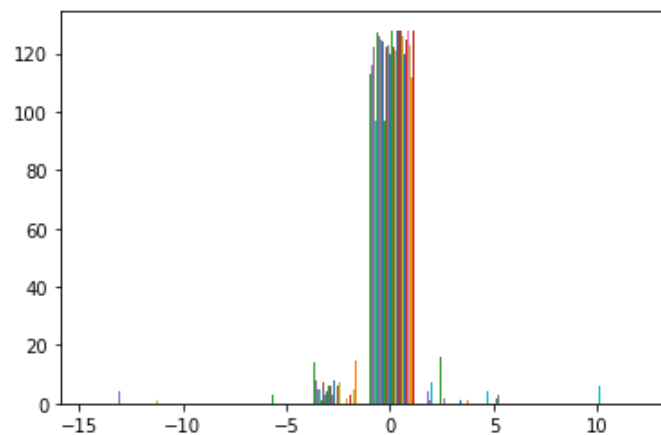
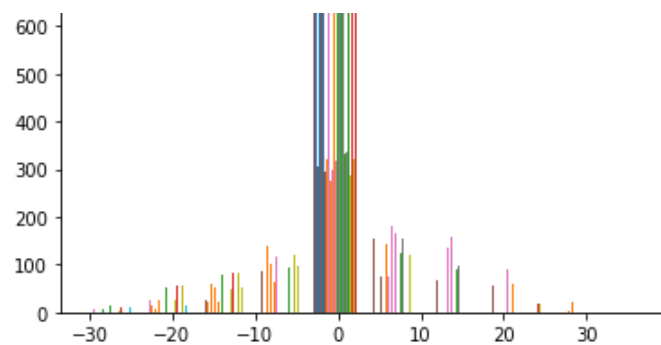
```
Epoch 5/50: loss = 1.6021393644126982
Epoch 10/50: loss = 1.5985715699298186
Epoch 15/50: loss = 1.5624256906820824
Epoch 20/50: loss = 1.593509593272557
Epoch 25/50: loss = 1.5915735457146003
Epoch 30/50: loss = 1.608829267389039
Epoch 35/50: loss = 1.62321705817252
Epoch 40/50: loss = 1.6164474601976628
Epoch 45/50: loss = 1.6298084917409386
Epoch 50/50: loss = 1.5982098766245372
```

```
1 model.loss_curve()
```



```
1 for layer in model.layers:
2     if type(layer) == Linear:
3         plt.hist(layer.Ws)
4         plt.show()
```





✓ 0s completed at 3:01 PM

