

Name: **Amrit Parimi** UNI: **ap4142**

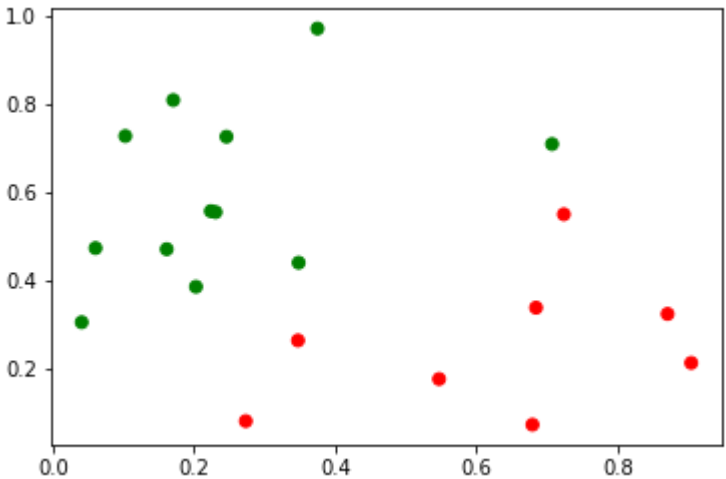
##Problem 1

```
In [1]: 1 import numpy as np
        2 import matplotlib.pyplot as plt
        3
```

```
In [303]: 1
          2 a,b = np.random.rand(20),np.random.rand(20)
          3 x_train = [[a[i],b[i]] for i in range(20)]
          4 y_train = []
          5 for i in range(20):
          6     if(x_train[i][0]>x_train[i][1]):
          7         y_train.append(1.0)
          8     else:
          9         y_train.append(-1.0)
         10
         11
```

```
In [279]: 1 a,b = np.random.rand(1000),np.random.rand(1000)
          2 x_test = [[a[i],b[i]] for i in range(1000)]
          3 y_test = []
          4 for i in range(1000):
          5     if(x_test[i][0]>x_test[i][1]):
          6         y_test.append(1.0)
          7     else:
          8         y_test.append(-1.0)
```

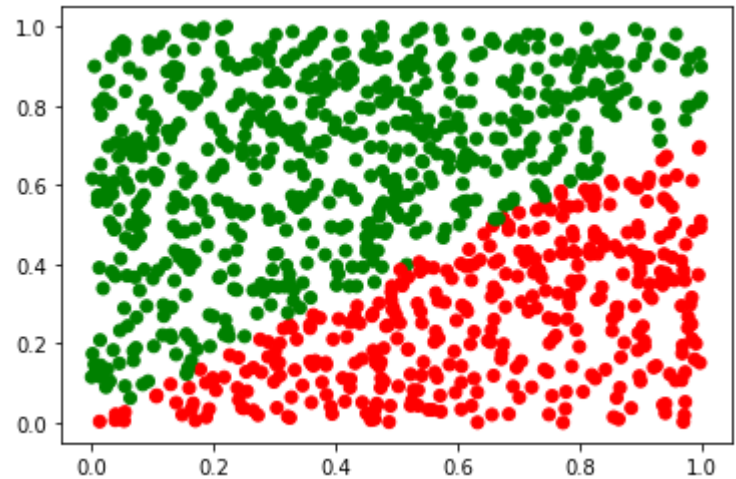
```
In [304]: 1 def predict(w,x):
          2     return 1.0 if w[0]*x[0]+w[1]*x[1]>=0 else -1.0
          3
          4 def percept_train(x_train, a, lr, epochs):
          5     w = [0,0]
          6     for ep in range(epochs):
          7         err = 0.0
          8         for i in range(len(x_train)):
          9             y_pred = predict(w,x_train[i])
         10             # error = max(0,a - y_train[i]*(w[0]*x_train[i][0]+w[1]*x_train[
         11             # print(a - y_train[i]*(w[0]*x_train[i][0]+w[1]*x_train[i][1]))
         12             if(y_train[i]*(w[0]*x_train[i][0]+w[1]*x_train[i][1])<=a):
         13                 w[0]+=lr*x_train[i][0]*y_train[i]
         14                 w[1]+=lr*x_train[i][1]*y_train[i]
         15             # print('>epoch=%d, Lrate=%.3f, error=%.3f' % (ep, lr, err))
         16         return w
         17
         18 plt.scatter([x_train[i][0] for i in range(20)], [x_train[i][1] for i in range
         19 plt.show()
```



In [305]:

```
1 # 1.1
2 w = percept_train(x_train,0,0.1,50)
3 print(w)
4
5 acc = 0
6 y_pred = []
7 # print(predict(w,x_test[0]),w[0]*x_test[0][0]+w[1]*x_test[0][1])
8 for i in range(len(x_test)):
9     y_pred.append(predict(w,x_test[i]))
10    if(y_pred[-1]==y_test[i]):
11        acc+=1
12 acc = acc/len(x_test)
13 print(acc)
14 plt.scatter([x_test[i][0] for i in range(1000)], [x_test[i][1] for i in range
15 plt.show()
```

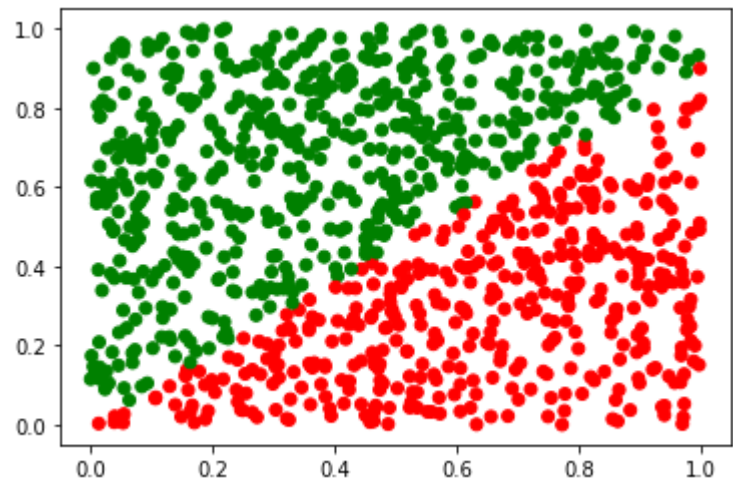
[0.06943906675610265, -0.09107289548530381]
0.902



In [306]:

```
1 # 1.2
2 w = percept_train(x_train,1,0.1,200)
3 print(w)
4
5 acc = 0
6 y_pred = []
7 # print(predict(w,x_test[0]),w[0]*x_test[0][0]+w[1]*x_test[0][1])
8 for i in range(len(x_test)):
9     y_pred.append(predict(w,x_test[i]))
10    if(y_pred[-1]==y_test[i]):
11        acc+=1
12 acc = acc/len(x_test)
13 print(acc)
14 plt.scatter([x_test[i][0] for i in range(1000)], [x_test[i][1] for i in range
15 plt.show()
```

[7.061454974712272, -7.848020193905932]
0.955



1.3 We can observe that we obtain better accuracy when we use hinge loss. We get better accuracy using hinge loss when compared to perceptron criterion because hinge loss handles points closer to the boundary and normally misclassified points better. Perceptron loss gives a non-zero gradient only when there is a misclassification but Hinge loss gives non-zero gradient for misclassified points as well as the points very close to the boundary ($y(w \cdot x) < 1$) even though they are correctly classified. This makes the classifier learn and perform better as it has a more precise boundary. \

1.4 The classification should not change significantly in the case of Hinge loss because hinge loss trains with points close to boundary along with misclassified points making it more stable. We would therefore expect it to learn better than the perceptron and thus the classification of the test points should not change significantly.

In [6]:

```
1
```

##Problem 2

Problem 2.1 \ We typically train models with gradient descent using a loss function in terms of the weight. The derivative is then back-propagated to the previous layers of the model. The weight update of the previous layers is proportional to the gradient value at this layer. As we keep going towards the initial layers of a deep network, the value is exponentially proportional to te gradient value at this layer. As we use activation functions, in the process of back-propagation there is a chance for the value of the gradient to become 0 or very close to 0. Due to this the weight update of the previous layers is also affected as it is propotional. \ $\frac{d(tanh(x))}{dx} - > 0$ when x is either too high or too low. Similarly $\frac{d(sigmoid(x))}{dx} - > 0$ when x is either too high or too low. \ Below are the violin plots showing that the activations are more dense at value=1,-1 for tanh and 0,1 for sigmoid where the gradient is close to 0.

In [146]:

```

1 import keras
2 from keras.models import Sequential
3 from keras.layers import Conv2D, MaxPooling2D, Dense, Dropout, Flatten
4 from keras import backend as K
5 from tensorflow.keras import optimizers
6 import tensorflow as tf
7
8 from matplotlib import pyplot as plt
9 from matplotlib import rcParamsDefault
10
11
12 def grid_axes_it(n_plots, n_cols=3, enumerate=False, fig=None):
13     """
14     Iterate through Axes objects on a grid with n_cols columns and as many
15     rows as needed to accommodate n_plots many plots.
16     Args:
17         n_plots: Number of plots to plot onto figure.
18         n_cols: Number of columns to divide the figure into.
19         fig: Optional figure reference.
20     Yields:
21         n_plots many Axes objects on a grid.
22     """
23     n_rows = n_plots / n_cols + int(n_plots % n_cols > 0)
24
25     if not fig:
26         default_figsize = rcParamsDefault['figure.figsize']
27         fig = plt.figure(figsize=(
28             default_figsize[0] * n_cols,
29             default_figsize[1] * n_rows
30         ))
31
32     for i in range(1, n_plots + 1):
33         ax = plt.subplot(n_rows, n_cols, i)
34         yield ax
35
36
37 def create_mlp_model(
38     n_hidden_layers,
39     dim_layer,
40     input_shape,
41     n_classes,
42     kernel_initializer,
43     bias_initializer,
44     activation,
45 ):
46     """Create Multi-Layer Perceptron with given parameters."""
47     model = Sequential()
48     model.add(Dense(dim_layer, input_shape=input_shape, kernel_initializer=k
49                     bias_initializer=bias_initializer))
50     for i in range(n_hidden_layers):
51         model.add(Dense(dim_layer, activation=activation, kernel_initializer
52                         bias_initializer=bias_initializer))
53     model.add(Dense(n_classes, activation='softmax', kernel_initializer=kern
54                     bias_initializer=bias_initializer))
55     return model
56
57
58 def create_cnn_model(input_shape, num_classes, kernel_initializer='glorot_un
59                     bias_initializer='zeros'):
60     """Create CNN model similar to
61         https://github.com/keras-team/keras/blob/master/examples/mnist\_cnn.py
62     model = Sequential()
63     model.add(Conv2D(32, kernel_size=(3, 3),
64                     activation='relu',
65                     input_shape=input_shape,
66                     kernel_initializer=kernel_initializer,
67                     bias_initializer=bias_initializer))
68     model.add(Conv2D(64, (3, 3), activation='relu',
69                     kernel_initializer=kernel_initializer,
70                     bias_initializer=bias_initializer))
71     model.add(MaxPooling2D(pool_size=(2, 2)))
72     model.add(Dropout(0.25))
73     model.add(Flatten())

```

```

74     model.add(Dense(128, activation='relu',
75                     kernel_initializer=kernel_initializer,
76                     bias_initializer=bias_initializer))
77     model.add(Dropout(0.5))
78     model.add(Dense(num_classes, activation='softmax',
79                     kernel_initializer=kernel_initializer,
80                     bias_initializer=bias_initializer))
81     return model
82
83
84 def compile_model(model):
85     model.compile(loss=keras.losses.categorical_crossentropy,
86                  optimizer=optimizers.RMSprop(),
87                  metrics=['accuracy'])
88     return model
89
90
91 def get_init_id(init):
92     """
93     Returns string ID summarizing initialization scheme and its parameters.
94     Args:
95         init: Instance of some initializer from keras.initializers.
96     """
97     try:
98         init_name = str(init).split('.')[2].split(' ')[0]
99     except:
100         init_name = str(init).split(' ')[0].replace('.', '_')
101
102     param_list = []
103     config = init.get_config()
104     for k, v in config.items():
105         if k == 'seed':
106             continue
107         param_list.append('{k}-{v}'.format(k=k, v=v))
108     init_params = '__'.join(param_list)
109
110     return '|'.join([init_name, init_params])
111
112
113 def get_activations(model, x, mode=0.0):
114     """Extract activations with given model and input vector x."""
115     outputs = [layer.output for layer in model.layers]
116     activations = K.function([model.input], outputs)
117     output_elts = activations(x)
118     return output_elts
119
120
121 class LossHistory(keras.callbacks.Callback):
122     """A custom keras callback for recording losses during network training.
123
124     def on_train_begin(self, logs={}):
125         self.losses = []
126         self.epoch_losses = []
127         self.epoch_val_losses = []
128
129     def on_batch_end(self, batch, logs={}):
130         self.losses.append(logs.get('loss'))
131
132     def on_epoch_end(self, epoch, logs={}):
133         self.epoch_losses.append(logs.get('loss'))
134         self.epoch_val_losses.append(logs.get('val_loss'))

```

In [308]:

```

1  import keras
2  import matplotlib.pyplot as plt
3  import numpy as np
4  import pandas as pd
5  import seaborn as sns
6  from keras import initializers
7  from keras.datasets import mnist
8  from tensorflow.keras import optimizers
9
10
11  seed = 10
12
13  # Number of points to plot
14  n_train = 1000
15  n_test = 100
16  n_classes = 10
17
18  # Network params
19  n_hidden_layers = 5
20  dim_layer = 100
21  batch_size = n_train
22  epochs = 1
23
24  # Load and prepare MNIST dataset.
25  n_train = 60000
26  n_test = 10000
27
28  (x_train, y_train), (x_test, y_test) = mnist.load_data()
29  num_classes = len(np.unique(y_test))
30  data_dim = 28 * 28
31
32  x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
33  x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
34  x_train /= 255
35  x_test /= 255
36
37  y_train = keras.utils.all_utils.to_categorical(y_train, num_classes)
38  y_test = keras.utils.all_utils.to_categorical(y_test, num_classes)
39
40  # Run the data through a few MLP models and save the activations from
41  # each layer into a Pandas DataFrame.
42  rows = []
43  sigmas = [0.10, 0.14, 0.28]
44  for stddev in sigmas:
45      init = initializers.RandomNormal(mean=0.0, stddev=stddev, seed=seed)
46      activation = 'tanh'
47
48      model = create_mlp_model(
49          n_hidden_layers,
50          dim_layer,
51          (data_dim,),
52          n_classes,
53          init,
54          'zeros',
55          activation
56      )
57      compile_model(model)
58      output_elts = get_activations(model, x_test)
59      n_layers = len(model.layers)
60      i_output_layer = n_layers - 1
61
62      for i, out in enumerate(output_elts[:-1]):
63          if i > 0 and i != i_output_layer:
64              for out_i in out.ravel()[::20]:
65                  rows.append([i, stddev, out_i])
66
67  df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviation', 'Output'])
68
69  # Plot previously saved activations from the 5 hidden layers
70  # using different initialization schemes.
71  fig = plt.figure(figsize=(12, 6))
72  axes = grid_axes_it(len(sigmas), 1, fig=fig)
73  for sig in sigmas:

```

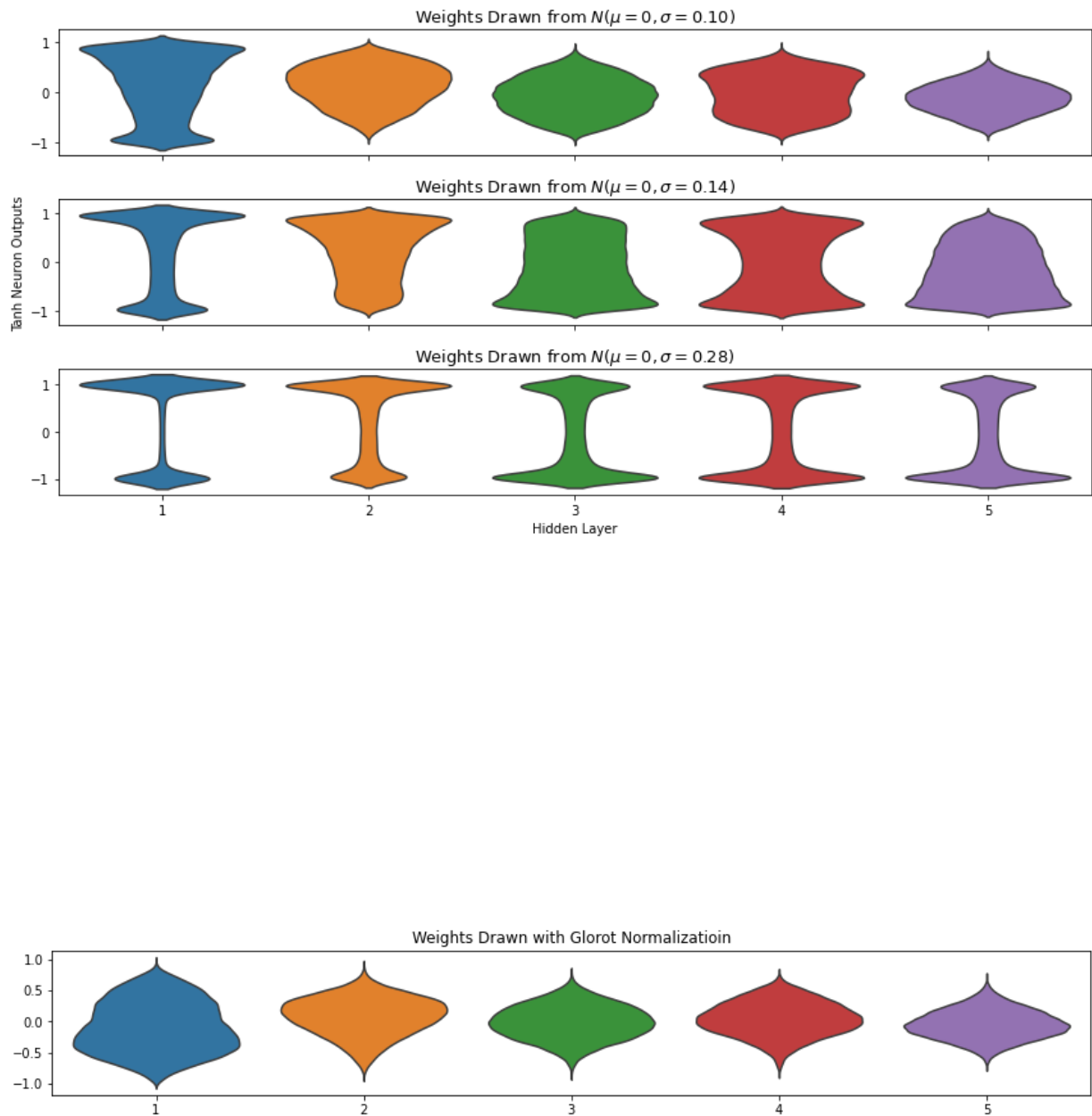
```

74     ax = next(axes)
75     ddf = df[df['Standard Deviation'] == sig]
76     sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='cou
77
78     ax.set_xlabel('')
79     ax.set_ylabel('')
80
81     ax.set_title('Weights Drawn from $N(\mu = 0, \sigma = \{%.2f\})$' % sig, f
82
83     if sig == sigmas[1]:
84         ax.set_ylabel("Tanh Neuron Outputs")
85     if sig != sigmas[-1]:
86         ax.set_xticklabels(())
87     else:
88         ax.set_xlabel("Hidden Layer")
89
90 plt.tight_layout()
91 plt.show()
92
93
94
95
96
97
98
99 seed = 10
100
101 # Number of points to plot
102 n_train = 1000
103 n_test = 100
104 n_classes = 10
105
106 # Network params
107 n_hidden_layers = 5
108 dim_layer = 100
109 batch_size = n_train
110 epochs = 1
111
112 # Load and prepare MNIST dataset.
113 n_train = 60000
114 n_test = 10000
115
116 (x_train, y_train), (x_test, y_test) = mnist.load_data()
117 num_classes = len(np.unique(y_test))
118 data_dim = 28 * 28
119
120 x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
121 x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
122 x_train /= 255
123 x_test /= 255
124
125 y_train = keras.utils.all_utils.to_categorical(y_train, num_classes)
126 y_test = keras.utils.all_utils.to_categorical(y_test, num_classes)
127
128 # Run the data through a few MLP models and save the activations from
129 # each layer into a Pandas DataFrame.
130 rows = []
131 # sigmas = [0.10, 0.14, 0.28]
132 # for stddev in sigmas:
133 init = initializers.GlorotNormal(seed=seed)
134 activation = 'tanh'
135
136 model = create_mlp_model(
137     n_hidden_layers,
138     dim_layer,
139     (data_dim,),
140     n_classes,
141     init,
142     'zeros',
143     activation
144 )
145 compile_model(model)
146 output_elts = get_activations(model, x_test)
147 n_layers = len(model.layers)

```



```
148 i_output_layer = n_layers - 1
149
150 for i, out in enumerate(output_elts[:-1]):
151     if i > 0 and i != i_output_layer:
152         for out_i in out.ravel()[::20]:
153             rows.append([i, stddev, out_i])
154
155 df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviation', 'Output'])
156
157 # Plot previously saved activations from the 5 hidden layers
158 # using different initialization schemes.
159 fig = plt.figure(figsize=(12, 6))
160 axes = grid_axes_it(len(sigmas), 1, fig=fig)
161 # for sig in sigmas:
162 ax = next(axes)
163 ddf = df[df['Standard Deviation'] == sig]
164 sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count',
165
166 ax.set_xlabel('')
167 ax.set_ylabel('')
168
169 ax.set_title('Weights Drawn with Glorot Normalization')
170
171 plt.tight_layout()
172 plt.show()
```



Above are the graphs showing how the activations in the initial are concentrated at value=1,-1 in other words, gradients are close to 0 proving gradient vanishing. \ We can also observe that for higher standard deviations there is a higher chance of gradient vanishing. \ We can also observe how using Glorot normalization has prevented the problem of vanishing gradient as the Tanh activations are well distributed.

In [147]:

```

1 import keras
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import pandas as pd
5 import seaborn as sns
6 from keras import initializers
7 from keras.datasets import mnist
8 from tensorflow.keras import optimizers
9
10
11 seed = 10
12
13 # Number of points to plot
14 n_train = 1000
15 n_test = 100
16 n_classes = 10
17
18 # Network params
19 n_hidden_layers = 5
20 dim_layer = 100
21 batch_size = n_train
22 epochs = 1
23
24 # Load and prepare MNIST dataset.
25 n_train = 60000
26 n_test = 10000
27
28 (x_train, y_train), (x_test, y_test) = mnist.load_data()
29 num_classes = len(np.unique(y_test))
30 data_dim = 28 * 28
31
32 x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
33 x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
34 x_train /= 255
35 x_test /= 255
36
37 y_train = keras.utils.all_utils.to_categorical(y_train, num_classes)
38 y_test = keras.utils.all_utils.to_categorical(y_test, num_classes)
39
40 # Run the data through a few MLP models and save the activations from
41 # each layer into a Pandas DataFrame.
42 rows = []
43 sigmas = [0.10, 0.14, 0.28]
44 for stddev in sigmas:
45     init = initializers.RandomNormal(mean=0.0, stddev=stddev, seed=seed)
46     activation = 'sigmoid'
47
48     model = create_mlp_model(
49         n_hidden_layers,
50         dim_layer,
51         (data_dim,),
52         n_classes,
53         init,
54         'zeros',
55         activation
56     )
57     compile_model(model)
58     output_elts = get_activations(model, x_test)
59     n_layers = len(model.layers)
60     i_output_layer = n_layers - 1
61
62     for i, out in enumerate(output_elts[:-1]):
63         if i > 0 and i != i_output_layer:
64             for out_i in out.ravel()[::20]:
65                 rows.append([i, stddev, out_i])
66
67 df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviation', 'Output'])
68
69 # Plot previously saved activations from the 5 hidden layers
70 # using different initialization schemes.
71 fig = plt.figure(figsize=(12, 6))
72 axes = grid_axes_it(len(sigmas), 1, fig=fig)
73 for sig in sigmas:

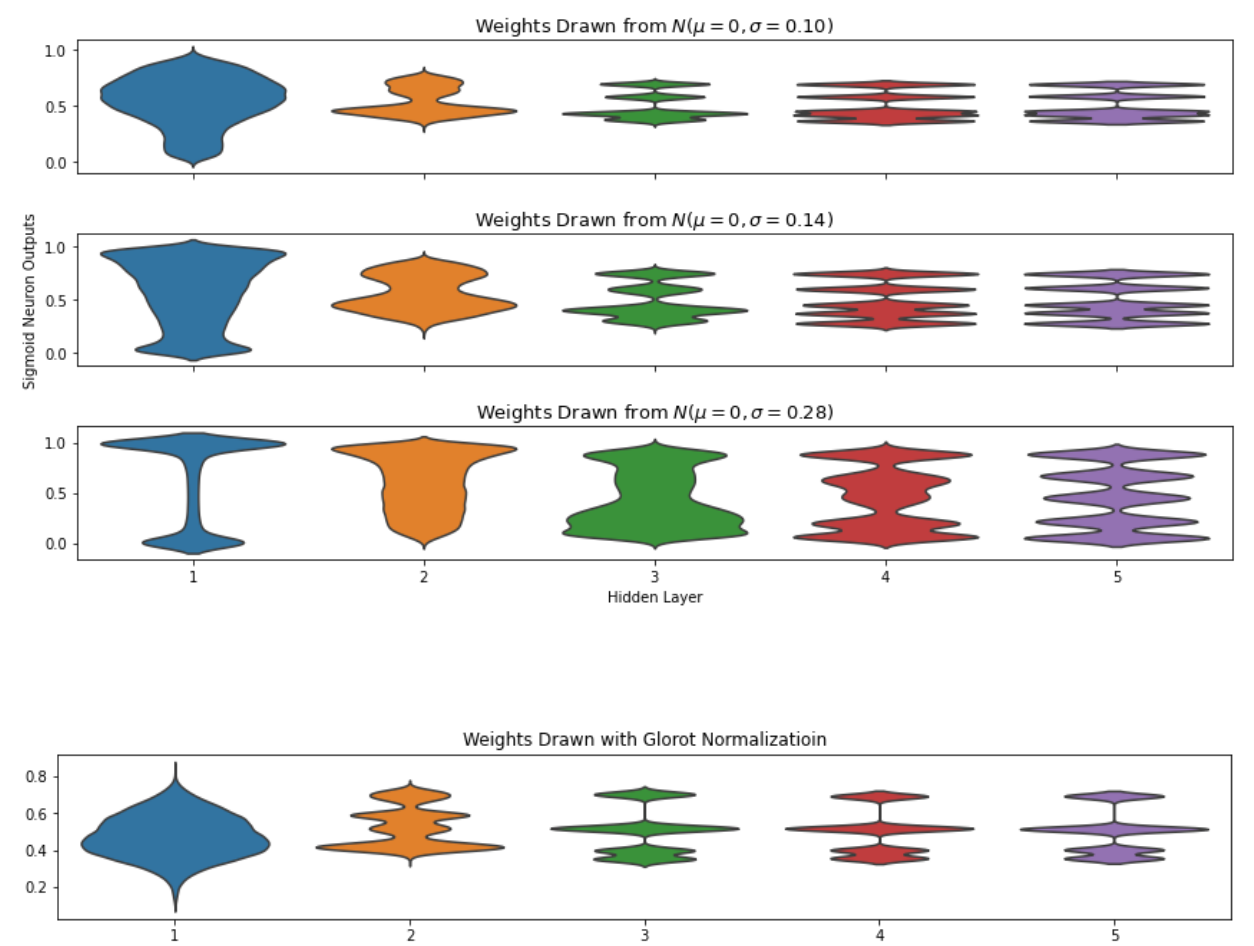
```

```

74     ax = next(axes)
75     ddf = df[df['Standard Deviation'] == sig]
76     sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='cou
77
78     ax.set_xlabel('')
79     ax.set_ylabel('')
80
81     ax.set_title('Weights Drawn from $N(\mu = 0, \sigma = \{%.2f\})$' % sig, f
82
83     if sig == sigmas[1]:
84         ax.set_ylabel("Sigmoid Neuron Outputs")
85     if sig != sigmas[-1]:
86         ax.set_xticklabels(())
87     else:
88         ax.set_xlabel("Hidden Layer")
89
90 plt.tight_layout()
91 plt.show()
92
93
94
95
96
97 import keras
98 import matplotlib.pyplot as plt
99 import numpy as np
100 import pandas as pd
101 import seaborn as sns
102 from keras import initializers
103 from keras.datasets import mnist
104 from tensorflow.keras import optimizers
105
106
107 seed = 10
108
109 # Number of points to plot
110 n_train = 1000
111 n_test = 100
112 n_classes = 10
113
114 # Network params
115 n_hidden_layers = 5
116 dim_layer = 100
117 batch_size = n_train
118 epochs = 1
119
120 # Load and prepare MNIST dataset.
121 n_train = 60000
122 n_test = 10000
123
124 (x_train, y_train), (x_test, y_test) = mnist.load_data()
125 num_classes = len(np.unique(y_test))
126 data_dim = 28 * 28
127
128 x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
129 x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
130 x_train /= 255
131 x_test /= 255
132
133 y_train = keras.utils.all_utils.to_categorical(y_train, num_classes)
134 y_test = keras.utils.all_utils.to_categorical(y_test, num_classes)
135
136 # Run the data through a few MLP models and save the activations from
137 # each Layer into a Pandas DataFrame.
138 rows = []
139 # sigmas = [0.10, 0.14, 0.28]
140 # for stddev in sigmas:
141 init = initializers.GlorotNormal(seed=seed)
142 activation = 'sigmoid'
143
144 model = create_mlp_model(
145     n_hidden_layers,
146     dim_layer,
147     (data_dim,),

```

```
148     n_classes,
149     init,
150     'zeros',
151     activation
152 )
153 compile_model(model)
154 output_elts = get_activations(model, x_test)
155 n_layers = len(model.layers)
156 i_output_layer = n_layers - 1
157
158 for i, out in enumerate(output_elts[:-1]):
159     if i > 0 and i != i_output_layer:
160         for out_i in out.ravel()[::20]:
161             rows.append([i, stddev, out_i])
162
163 df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviation', 'Output'])
164
165 # Plot previously saved activations from the 5 hidden layers
166 # using different initialization schemes.
167 fig = plt.figure(figsize=(12, 6))
168 axes = grid_axes_it(len(sigmas), 1, fig=fig)
169 # for sig in sigmas:
170 ax = next(axes)
171 ddf = df[df['Standard Deviation'] == sig]
172 sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count',
173
174
175 ax.set_xlabel('')
176 ax.set_ylabel('')
177 ax.set_title('Weights Drawn with Glorot Normalization')
178
179 plt.tight_layout()
180 plt.show()
```



Above are the graphs showing how the activations in the initial layers with are concentrated at value=1,0 in other words, gradients close to 0 when we use high standard deviation. \ We can also observe that for higher standard deviations there is a higher chance of gradient vanishing. \ We can also observe how using Glorot normalization has prevented the problem of vanishing gradient as the Sigmoid activations are well distributed.

In [310]:

```

1  import keras
2  import matplotlib.pyplot as plt
3  import numpy as np
4  import pandas as pd
5  import seaborn as sns
6  from keras import initializers
7  from keras.datasets import mnist
8  from tensorflow.keras import optimizers
9
10
11
12  seed = 10
13
14  # Number of points to plot
15  n_train = 1000
16  n_test = 100
17  n_classes = 10
18
19  # Network params
20  n_hidden_layers = 5
21  dim_layer = 100
22  batch_size = n_train
23  epochs = 1
24
25  # Load and prepare MNIST dataset.
26  n_train = 60000
27  n_test = 10000
28
29  (x_train, y_train), (x_test, y_test) = mnist.load_data()
30  num_classes = len(np.unique(y_test))
31  data_dim = 28 * 28
32
33  x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
34  x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
35  x_train /= 255
36  x_test /= 255
37
38  y_train = keras.utils.all_utils.to_categorical(y_train, num_classes)
39  y_test = keras.utils.all_utils.to_categorical(y_test, num_classes)
40
41  # Run the data through a few MLP models and save the activations from
42  # each layer into a Pandas DataFrame.
43  rows = []
44  # sigmas = [0.10, 0.14, 0.28]
45  # for stddev in sigmas:
46  init = initializers.GlorotNormal(seed=seed)
47  activation = 'relu'
48
49  model = create_mlp_model(
50      n_hidden_layers,
51      dim_layer,
52      (data_dim,),
53      n_classes,
54      init,
55      'zeros',
56      activation
57  )
58  compile_model(model)
59  output_elts = get_activations(model, x_test)
60  n_layers = len(model.layers)
61  i_output_layer = n_layers - 1
62
63  for i, out in enumerate(output_elts[:-1]):
64      if i > 0 and i != i_output_layer:
65          for out_i in out.ravel()[::20]:
66              rows.append([i, stddev, out_i])
67
68  df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviation', 'Output'])
69
70  # Plot previously saved activations from the 5 hidden layers
71  # using different initialization schemes.
72  fig = plt.figure(figsize=(12, 6))
73  axes = grid_axes_it(len(sigmas), 1, fig=fig)

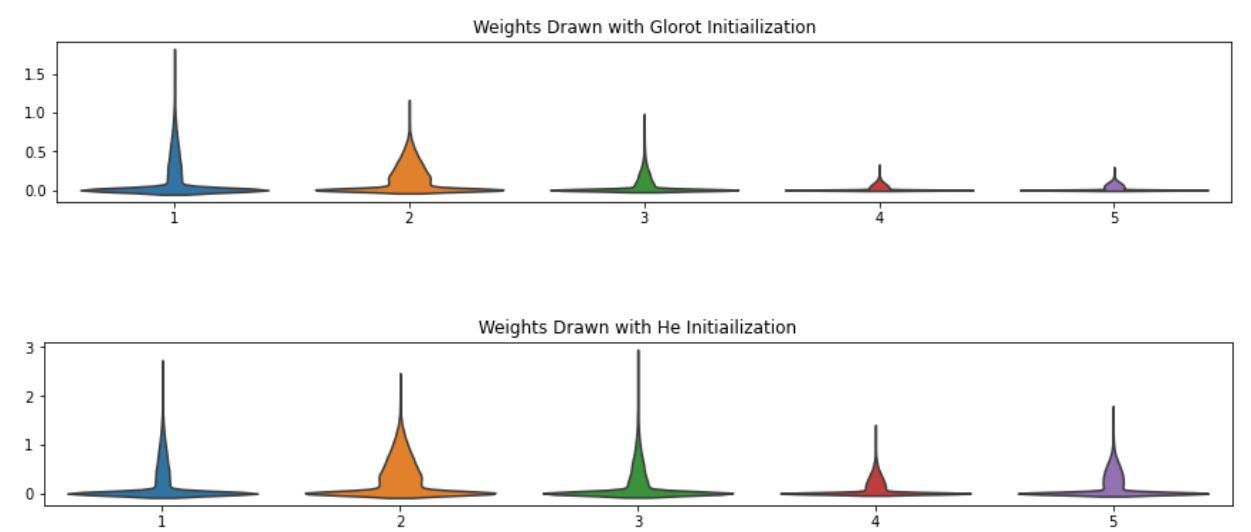
```

```

74 # for sig in sigmas:
75 ax = next(axes)
76 ddf = df[df['Standard Deviation'] == sig]
77 sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count',
78
79 ax.set_xlabel('')
80 ax.set_ylabel('')
81
82 ax.set_title('Weights Drawn with Glorot Initiaailization')
83
84 plt.tight_layout()
85 plt.show()
86
87
88
89
90
91
92
93 import keras
94 import matplotlib.pyplot as plt
95 import numpy as np
96 import pandas as pd
97 import seaborn as sns
98 from keras import initializers
99 from keras.datasets import mnist
100 from tensorflow.keras import optimizers
101
102
103 seed = 10
104
105 # Number of points to plot
106 n_train = 1000
107 n_test = 100
108 n_classes = 10
109
110 # Network params
111 n_hidden_layers = 5
112 dim_layer = 100
113 batch_size = n_train
114 epochs = 1
115
116 # Load and prepare MNIST dataset.
117 n_train = 60000
118 n_test = 10000
119
120 (x_train, y_train), (x_test, y_test) = mnist.load_data()
121 num_classes = len(np.unique(y_test))
122 data_dim = 28 * 28
123
124 x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
125 x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
126 x_train /= 255
127 x_test /= 255
128
129 y_train = keras.utils.all_utils.to_categorical(y_train, num_classes)
130 y_test = keras.utils.all_utils.to_categorical(y_test, num_classes)
131
132 # Run the data through a few MLP models and save the activations from
133 # each layer into a Pandas DataFrame.
134 rows = []
135 # sigmas = [0.10, 0.14, 0.28]
136 # for stddev in sigmas:
137 init = initializers.HeNormal(seed=seed)
138 activation = 'relu'
139
140 model = create_mlp_model(
141     n_hidden_layers,
142     dim_layer,
143     (data_dim,),
144     n_classes,
145     init,
146     'zeros',
147     activation

```

```
148 )
149 compile_model(model)
150 output_elts = get_activations(model, x_test)
151 n_layers = len(model.layers)
152 i_output_layer = n_layers - 1
153
154 for i, out in enumerate(output_elts[:-1]):
155     if i > 0 and i != i_output_layer:
156         for out_i in out.ravel()[::20]:
157             rows.append([i, stddev, out_i])
158
159 df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviation', 'Output'])
160
161 # Plot previously saved activations from the 5 hidden layers
162 # using different initialization schemes.
163 fig = plt.figure(figsize=(12, 6))
164 axes = grid_axes_it(len(sigmas), 1, fig=fig)
165 # for sig in sigmas:
166 ax = next(axes)
167 ddf = df[df['Standard Deviation'] == sig]
168 sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count',
169
170 ax.set_xlabel('')
171 ax.set_ylabel('')
172
173 ax.set_title('Weights Drawn with He Initiaailization')
174
175 plt.tight_layout()
176 plt.show()
```



From the violin plot above, we can observe that the distribution of the activations of the 4th and 5th layers are concentrated at a very small value(<0.5) in the case of Xavier/Glorot initialization(may cause vanishing gradients) when compared to He initialization. Therefore He Initialization works better for ReLU activation.

```
In [10]: 1
```

```
In [10]: 1
```

Problem 2.2

In [148]:

```
1 def create_mlp_model(  
2     n_hidden_layers,  
3     dim_layer,  
4     input_shape,  
5     n_classes,  
6     kernel_initializer,  
7     bias_initializer,  
8     activation,  
9 ):  
10     """Create Multi-Layer Perceptron with given parameters."""  
11     model = Sequential()  
12     model.add(Dense(dim_layer, input_shape=input_shape, kernel_initializer=k  
13                     bias_initializer=bias_initializer))  
14     for i in range(n_hidden_layers):  
15         model.add(Dense(dim_layer, activation=activation, kernel_initializer  
16                         bias_initializer=bias_initializer))  
17     model.add(Dense(n_classes, activation='linear', kernel_initializer=kerne  
18                     bias_initializer=bias_initializer))  
19     return model
```

1.1 The function I used show the phenomenon of dead ReLU network is $f(x) = |x|$. \ I have classified a network as a dying ReLU network using the fact that it gives the same answer as output irrespective of the input(test set).

In [329]:

```
1 minibatch = 64  
2 runs=1000  
3 c=0  
4 for _ in range(runs):  
5     x_train = np.random.uniform(-np.sqrt(7),np.sqrt(7),3000)  
6     y_train = abs(x_train)  
7     x_test = np.random.uniform(-np.sqrt(7),np.sqrt(7),100)  
8     model = create_mlp_model(10,2,(1,),1,kernel_initializer=initializers.Ran  
9     model.compile(optimizer='adam',loss=tf.keras.losses.MeanSquaredError())  
10    model.fit(x_train,y_train,batch_size=minibatch,epochs=20,verbose=0)  
11    y_pred = model.predict(x_test)  
12    # print(tf.keras.losses.MeanSquaredError()(y_test,y_pred))  
13    if(len(set(y_pred[:,0]))==1):  
14        c+=1  
15    # print(c/(_+1))  
16 print('Network Collapse using ReLU ={}%'.format(c*100/runs))
```

Network Collapse using ReLU =99.6%

In the case of ReLU the percentage of dead networks is = 99.6% \ This is higher than the percentage reported in Lu et al.

In [151]:

```
1 minibatch = 64  
2 runs=1000  
3 c=0  
4 for _ in range(runs):  
5     x_train = np.random.uniform(-np.sqrt(7),np.sqrt(7),3000)  
6     y_train = abs(x_train*np.sin(5*x_train))  
7     x_test = np.random.uniform(-np.sqrt(7),np.sqrt(7),100)  
8     model = create_mlp_model(10,2,(1,),1,kernel_initializer=initializers.Ran  
9     model.compile(optimizer='adam',loss=tf.keras.losses.MeanSquaredError())  
10    model.fit(x_train,y_train,batch_size=minibatch,epochs=20,verbose=0)  
11    y_pred = model.predict(x_test)  
12    # print(tf.keras.losses.MeanSquaredError()(y_test,y_pred))  
13    if(len(set(y_pred[:,0]))==1):  
14        c+=1  
15 print('Network Collapse using Leaky ReLU ={}%'.format(c*100/runs))
```

Network Collapse using Leaky ReLU =83.0%

In the case of Leaky ReLU the percentage of dead networks is = 83% \ Yes Leaky ReLU helped in reducing the percentage of dying neurons as the gradient is non-zero even for negatie input values. This keeps the neurons active even though their value might be less.

In []:

1

In []:

1

In [167]:

1

Problem 3

3.1. **Co-adaptation:** In terms of neural networks, co-adaptation is the phenomenon of units highly depending on each other. In neural networks, the derivative received by each parameter is updated using the derivative with respect to loss and is back-propagated which is inturn dependent on other units. Although final loss function is reduced, units may change in a way that they fix up the mistakes of the other units. These are called co-adaptations. Theseco-adaptations may increase the train accuracy but may cause the network to overfit and fail at generalization.

Internal covariance shift: Internal covariance shift is defines as the We refer to the change in the distributions of internal nodes of a deep network, in the course of training. Training Deep Neural Networks is complicated because the distribution of each layer’s inputs changes during training as the parameters of the previous layers change. To make sure the model converges in spite of this, we require lower learning rates and careful parameter initialization. This makes it very hard to train models with saturating nonlinearities.

In [88]:

```
1  # 3.2
2  import time
3  from keras.datasets import mnist
4  from matplotlib import pyplot
5  import tensorflow as tf
6  import keras
7
8  num_classes = 10
9  input_shape = (28, 28, 1)
10
11 # the data, split between train and test sets
12 (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
13
14 # Scale images to the [0, 1] range
15 x_train = x_train.astype("float32") / 255
16 x_test = x_test.astype("float32") / 255
17 # Make sure images have shape (28, 28, 1)
18 x_train = np.expand_dims(x_train, -1)
19 x_test = np.expand_dims(x_test, -1)
20 print("x_train shape:", x_train.shape)
21 print(x_train.shape[0], "train samples")
22 print(x_test.shape[0], "test samples")
23
24
25 # convert class vectors to binary class matrices
26 y_train = keras.utils.all_utils.to_categorical(y_train, num_classes)
27 y_test = keras.utils.all_utils.to_categorical(y_test, num_classes)
```

x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples

```
In [372]: 1 def net():
2         return tf.keras.models.Sequential([
3             tf.keras.layers.Normalization(),
4             # tf.keras.layers.BatchNormalization(),
5             tf.keras.layers.Conv2D(filters=6, kernel_size=5, input_shape=(28, 28,
6             tf.keras.layers.BatchNormalization(),
7             tf.keras.layers.Activation('sigmoid'),
8             tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
9             tf.keras.layers.Conv2D(filters=16, kernel_size=5),
10            tf.keras.layers.BatchNormalization(),
11            tf.keras.layers.Activation('sigmoid'),
12            tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
13            tf.keras.layers.Flatten(),
14            tf.keras.layers.Dense(120),
15            tf.keras.layers.BatchNormalization(),
16            tf.keras.layers.Activation('sigmoid'),
17            tf.keras.layers.Dense(84),
18            tf.keras.layers.BatchNormalization(),
19            tf.keras.layers.Activation('sigmoid'),
20            tf.keras.layers.Dense(10, activation = 'softmax'),])
```

```
In [373]: 1 model = net()
2         model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc
3         model.fit(x_train, y_train, 128, 5, validation_split=0.1)
```

Epoch 1/5
422/422 [=====] - 31s 72ms/step - loss: 0.5432 - accur
acy: 0.9003 - val_loss: 2.4032 - val_accuracy: 0.2858
Epoch 2/5
422/422 [=====] - 30s 71ms/step - loss: 0.1335 - accur
acy: 0.9714 - val_loss: 0.4917 - val_accuracy: 0.8350
Epoch 3/5
422/422 [=====] - 30s 71ms/step - loss: 0.0851 - accur
acy: 0.9788 - val_loss: 0.1243 - val_accuracy: 0.9650
Epoch 4/5
422/422 [=====] - 30s 71ms/step - loss: 0.0658 - accur
acy: 0.9826 - val_loss: 0.3090 - val_accuracy: 0.9020
Epoch 5/5
422/422 [=====] - 30s 71ms/step - loss: 0.0566 - accur
acy: 0.9839 - val_loss: 0.1413 - val_accuracy: 0.9550

Out[373]: <keras.callbacks.History at 0x7ff281e20f10>

```
In [374]: 1 model.evaluate(x_test, y_test)
```

313/313 [=====] - 3s 9ms/step - loss: 0.1521 - accurac
y: 0.9511

Out[374]: [0.15214568376541138, 0.9510999917984009]

In [341]:

```
1 #Order of parameters of BatchNorm gamma, beta, running mean, std
2 batchnorm1 = []
3 i=0
4 for l in (model.layers):
5     if('batch_normalization' in str(l)):
6         i+=1
7         batchnorm1.append(l.get_weights())
8         print('The learned batch norm parameters of BatchNorm layer {} are {}'.format(l.name, batchnorm1[-1]))
9 print(len(batchnorm1))
```

The learned batch norm parameters of BatchNorm layer 1 are [array([1.0397717, 0.95228153, 1.0521868, 1.0007092, 0.9660674, 1.0419983], dtype=float32), array([-0.08222771, 0.02226105, -0.00756605, -0.09792377, -0.01121139, -0.01409738], dtype=float32), array([0.00677308, 0.07345571, -0.18209235, 0.0201771, -0.01562181, -0.10563064], dtype=float32), array([0.01389957, 0.02929497, 0.11323152, 0.02999649, 0.03728595, 0.06546629], dtype=float32)]

The learned batch norm parameters of BatchNorm layer 2 are [array([1.0126945, 1.0781525, 1.0164979, 0.9803727, 0.9678099, 0.99029547, 1.0196694, 1.0150005, 0.9890731, 1.0432907, 0.973346, 0.99708146, 0.94650483, 0.9571472, 0.97539544, 0.9821794], dtype=float32), array([0.01839233, 0.1196842, -0.03220168, -0.04213286, -0.04021622, 0.01078884, 0.00708483, 0.07468873, 0.00502367, 0.04023496, -0.04142677, -0.00916221, -0.01372475, -0.03107252, -0.02290029, -0.02308035], dtype=float32), array([-0.3081783, 0.82954377, 0.05706682, -0.11763691, 0.09120083, 0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000], dtype=float32)]

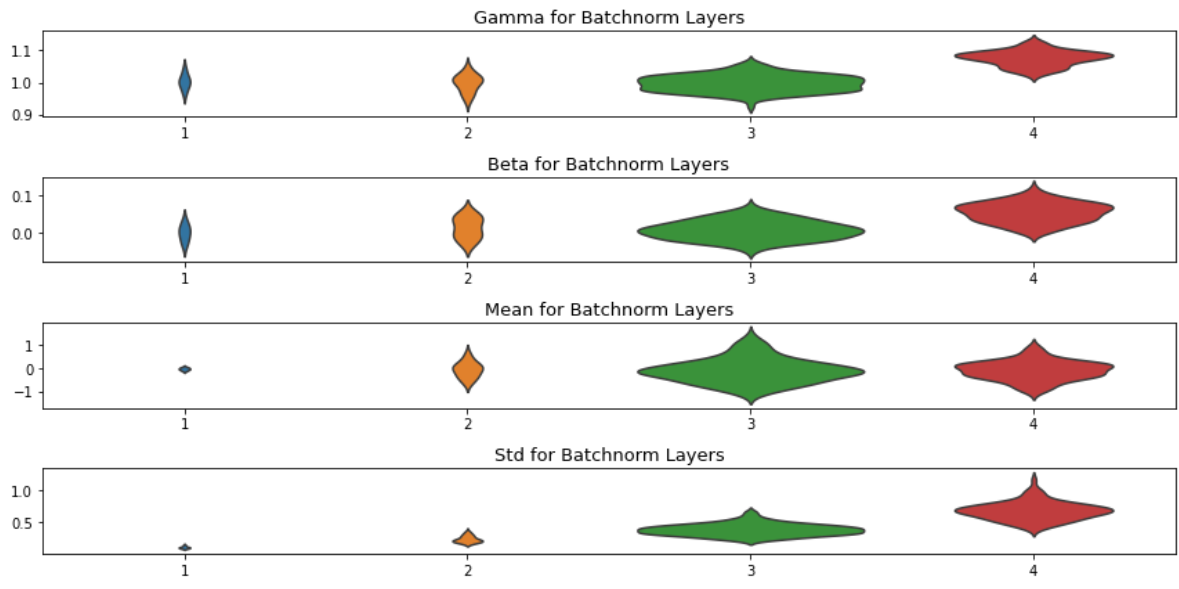
In [80]:

```
1 print(batchnorm1[0],batchnorm1[0][0])
2 d1 = {'x':[],'gamma':[]}#,'beta':[],'mean':[],'std':[]}
3 d2 = {'x':[],'beta':[]}
4 d3 = {'x':[],'mean':[]}
5 d4 = {'x':[],'std':[]}
6 lab = {'0':'gamma','1':'beta','2':'mean','3':'std'}
7 for k in range(len(batchnorm1)):
8     # for i in range(len(batchnorm1[k])):
9     # print(i,len(batchnorm1[k][0]))
10    for j in range(len(batchnorm1[k][0])):
11        d1['x'].append(k+1)
12        d1[lab[str(0)]].append(batchnorm1[k][0][j])
13    for j in range(len(batchnorm1[k][1])):
14        d2['x'].append(k+1)
15        d2[lab[str(1)]].append(batchnorm1[k][1][j])
16    for j in range(len(batchnorm1[k][2])):
17        d3['x'].append(k+1)
18        d3[lab[str(2)]].append(batchnorm1[k][2][j])
19    for j in range(len(batchnorm1[k][3])):
20        d4['x'].append(k+1)
21        d4[lab[str(3)]].append(batchnorm1[k][3][j])
22
23 # d = {'x':[1 for _ in range(len(batchnorm1[0][0]))].append([2 for _ in range(len(batchnorm1[0][1]))].append([3 for _ in range(len(batchnorm1[0][2]))].append([4 for _ in range(len(batchnorm1[0][3]))])
24 gamma = pd.DataFrame(data=d1)
25 beta = pd.DataFrame(data=d2)
26 mean = pd.DataFrame(data=d3)
27 std = pd.DataFrame(data=d4)
```

[array([0.99304897, 1.0106, 0.9962658, 0.9685931, 1.0104078, 1.0396566], dtype=float32), array([-0.01789312, 0.00611853, 0.01339216, 0.03063561, -0.03236492, -0.0023613], dtype=float32), array([-0.02392622, -0.11904573, 0.039437, 0.0100503, -0.06592131, -0.04163283], dtype=float32), array([0.11428282, 0.14178412, 0.11058541, 0.09558843, 0.09685789, 0.10353436], dtype=float32)] [0.99304897 1.0106 0.9962658 0.9685931 1.0104078 1.0396566]

In [87]:

```
1 fig = plt.figure(figsize=(12, 6))
2 axes = grid_axes_it(4, 1, fig=fig)
3 ax = next(axes)
4 # ddf = df[df['Standard Deviation'] == sig]
5 sns.violinplot(x='x', y='gamma', data=gamma, ax=ax, scale='count', inner=None)
6
7 # sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count')
8
9 ax.set_xlabel('')
10 ax.set_ylabel('')
11
12 ax.set_title('Gamma for Batchnorm Layers', fontsize=13)
13
14
15 ax = next(axes)
16 # ddf = df[df['Standard Deviation'] == sig]
17 sns.violinplot(x='x', y='beta', data=beta, ax=ax, scale='count', inner=None)
18
19 # sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count')
20
21 ax.set_xlabel('')
22 ax.set_ylabel('')
23
24 ax.set_title('Beta for Batchnorm Layers', fontsize=13)
25
26
27
28 ax = next(axes)
29 # ddf = df[df['Standard Deviation'] == sig]
30 sns.violinplot(x='x', y='mean', data=mean, ax=ax, scale='count', inner=None)
31
32 # sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count')
33
34 ax.set_xlabel('')
35 ax.set_ylabel('')
36
37 ax.set_title('Mean for Batchnorm Layers', fontsize=13)
38
39
40
41
42
43 ax = next(axes)
44 # ddf = df[df['Standard Deviation'] == sig]
45 sns.violinplot(x='x', y='std', data=std, ax=ax, scale='count', inner=None)
46
47 # sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count')
48
49 ax.set_xlabel('')
50 ax.set_ylabel('')
51
52 ax.set_title('Std for Batchnorm Layers', fontsize=13)
53
54
55
56
57
58 plt.tight_layout()
59 plt.show()
```



```
In [74]: 1
```

```
In [369]: 1 # 3.3
2 def net():
3     return tf.keras.models.Sequential([
4         tf.keras.layers.BatchNormalization(),
5         tf.keras.layers.Conv2D(filters=6, kernel_size=5, input_shape=(28, 28,
6         tf.keras.layers.BatchNormalization(),
7         tf.keras.layers.Activation('sigmoid'),
8         tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
9         tf.keras.layers.Conv2D(filters=16, kernel_size=5),
10        tf.keras.layers.BatchNormalization(),
11        tf.keras.layers.Activation('sigmoid'),
12        tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
13        tf.keras.layers.Flatten(),
14        tf.keras.layers.Dense(120),
15        tf.keras.layers.BatchNormalization(),
16        tf.keras.layers.Activation('sigmoid'),
17        tf.keras.layers.Dense(84),
18        tf.keras.layers.BatchNormalization(),
19        tf.keras.layers.Activation('sigmoid'),
20        tf.keras.layers.Dense(10, activation = 'softmax'),])
```

```
In [370]: 1 model = net()
2 model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc
3 model.fit(x_train, y_train, 128, 5, validation_split=0.1)
```

Epoch 1/5
422/422 [=====] - 36s 82ms/step - loss: 0.5366 - accuracy: 0.9100 - val_loss: 1.1415 - val_accuracy: 0.6973
Epoch 2/5
422/422 [=====] - 34s 82ms/step - loss: 0.1342 - accuracy: 0.9736 - val_loss: 0.2600 - val_accuracy: 0.9245
Epoch 3/5
422/422 [=====] - 34s 81ms/step - loss: 0.0865 - accuracy: 0.9792 - val_loss: 0.1320 - val_accuracy: 0.9598
Epoch 4/5
422/422 [=====] - 34s 82ms/step - loss: 0.0669 - accuracy: 0.9824 - val_loss: 0.0691 - val_accuracy: 0.9823
Epoch 5/5
422/422 [=====] - 34s 82ms/step - loss: 0.0561 - accuracy: 0.9847 - val_loss: 0.0682 - val_accuracy: 0.9808

Out[370]: <keras.callbacks.History at 0x7ff282b42250>

```
In [371]: 1 model.evaluate(x_test, y_test)
```

313/313 [=====] - 3s 9ms/step - loss: 0.0780 - accuracy: 0.9772

Out[371]: [0.07803839445114136, 0.9771999716758728]

In [346]:

```
1 #Order of parameters of BatchNorm gamma, beta, running mean, std
2 batchnorm1 = []
3 i=0
4 for l in (model.layers):
5     if('batch_normalization' in str(l)):
6         i+=1
7         batchnorm1.append(l.get_weights())
8         print('The learned batch norm parameters of BatchNorm layer {} are {}'.format(l.name, batchnorm1[-1]))
9 print(len(batchnorm1))
```

The learned batch norm parameters of BatchNorm layer 1 are [array([0.93576 9], dtype=float32), array([0.03544567], dtype=float32), array([0.13047461], dtype=float32), array([0.09480096], dtype=float32)]

The learned batch norm parameters of BatchNorm layer 2 are [array([0.9245941 6, 0.969346 , 1.0635519 , 1.0852412 , 0.9425522 , 1.079936], dtype=float32), array([-0.01662977, -0.06533749, -0.0796 4704, 0.0227954 , 0.05569836, -0.0122155], dtype=float32), array([-0.01847833, -0.00050582, -0.013 34626, -0.28055096, -0.07656857, -0.06905887], dtype=float32), array([0.36971968, 0.49936122, 0.252470 4 , 1.6194601 , 0.08810559, 0.47783393], dtype=float32)]

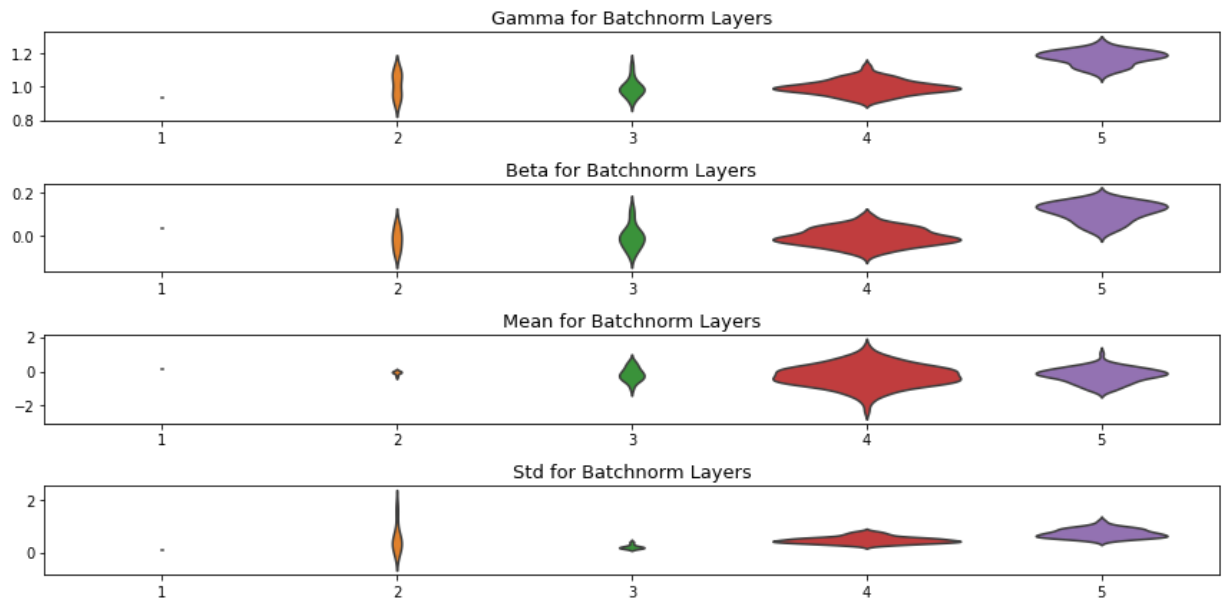
The learned batch norm parameters of BatchNorm layer 3 are [array([1.013412 , 0.93622553, 1.0098742 , 1.072019 , 0.98053354, 0.95102227, 0.9669562 , 0.9900335 , 0.96503115, 0.9981381 , 1.130085 , 0.9165584 , 0.9828812 , 1.0323027 , 0.99074376, 0.9763477], dtype=float32), array([-0.0166647 , -0.0360014 , 0.0102 2221, -0.06883993, -0.02338721, 0.00928411, 0.12379377, -0.08671878, -0.01531355, 0.0093356 , 0.0000000 , 0.0000000 , 0.0000000 , 0.0000000 , 0.0000000])]

In [349]:

```
1 d1 = {'x':[], 'gamma':[]}#, 'beta':[], 'mean':[], 'std':[]}
2 d2 = {'x':[], 'beta':[]}
3 d3 = {'x':[], 'mean':[]}
4 d4 = {'x':[], 'std':[]}
5 lab = {'0':'gamma', '1':'beta', '2':'mean', '3':'std'}
6 for k in range(len(batchnorm1)):
7     # for i in range(len(batchnorm1[k])):
8     # print(i, len(batchnorm1[k][0]))
9     for j in range(len(batchnorm1[k][0])):
10         d1['x'].append(k+1)
11         d1[lab[str(0)]].append(batchnorm1[k][0][j])
12     for j in range(len(batchnorm1[k][1])):
13         d2['x'].append(k+1)
14         d2[lab[str(1)]].append(batchnorm1[k][1][j])
15     for j in range(len(batchnorm1[k][2])):
16         d3['x'].append(k+1)
17         d3[lab[str(2)]].append(batchnorm1[k][2][j])
18     for j in range(len(batchnorm1[k][3])):
19         d4['x'].append(k+1)
20         d4[lab[str(3)]].append(batchnorm1[k][3][j])
21
22 gamma = pd.DataFrame(data=d1)
23 beta = pd.DataFrame(data=d2)
24 mean = pd.DataFrame(data=d3)
25 std = pd.DataFrame(data=d4)
```

In [350]:

```
1 fig = plt.figure(figsize=(12, 6))
2 axes = grid_axes_it(4, 1, fig=fig)
3 ax = next(axes)
4 # ddf = df[df['Standard Deviation'] == sig]
5 sns.violinplot(x='x', y='gamma', data=gamma, ax=ax, scale='count', inner=None)
6
7 # sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count')
8
9 ax.set_xlabel('')
10 ax.set_ylabel('')
11
12 ax.set_title('Gamma for Batchnorm Layers', fontsize=13)
13
14
15 ax = next(axes)
16 # ddf = df[df['Standard Deviation'] == sig]
17 sns.violinplot(x='x', y='beta', data=beta, ax=ax, scale='count', inner=None)
18
19 # sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count')
20
21 ax.set_xlabel('')
22 ax.set_ylabel('')
23
24 ax.set_title('Beta for Batchnorm Layers', fontsize=13)
25
26
27
28 ax = next(axes)
29 # ddf = df[df['Standard Deviation'] == sig]
30 sns.violinplot(x='x', y='mean', data=mean, ax=ax, scale='count', inner=None)
31
32 # sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count')
33
34 ax.set_xlabel('')
35 ax.set_ylabel('')
36
37 ax.set_title('Mean for Batchnorm Layers', fontsize=13)
38
39
40
41
42
43 ax = next(axes)
44 # ddf = df[df['Standard Deviation'] == sig]
45 sns.violinplot(x='x', y='std', data=std, ax=ax, scale='count', inner=None)
46
47 # sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, scale='count')
48
49 ax.set_xlabel('')
50 ax.set_ylabel('')
51
52 ax.set_title('Std for Batchnorm Layers', fontsize=13)
53
54
55
56
57
58 plt.tight_layout()
59 plt.show()
```

As we can observe using Batch Normalization instead of Standard Normalization has increased the test accuracy. \ Standard Norm - train loss: 0.0566, train accuracy: 0.9839; test loss: 0.1521, test accuracy: 0.9511 \ Batch Normalization - train loss: 0.0561, train accuracy: 0.9847; test loss: 0.0780, test accuracy: 0.9772 \ The train loss and accuracy is almost the same for both the models. Batch norm model's loss has very slightly higher train accuracy and lower train loss. \ The test accuracy of te Batch norm model is higher than that of the standard norm model. The test loss of tje Batchh norm model is lower than that of the standard norm model. \ Yes the batch normalization for the input layer has slightly improved the performance on the test set.

In []:

1

In [89]:

```
1 # Dropout
2
3 def net():
4     return tf.keras.models.Sequential([
5         tf.keras.layers.Dropout(0.2, noise_shape=None, seed=None),
6         tf.keras.layers.Conv2D(filters=6, kernel_size=5,input_shape=(28, 28,
7         tf.keras.layers.Activation('sigmoid'),
8         tf.keras.layers.Dropout(0.5, noise_shape=None, seed=None),
9         tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
10        tf.keras.layers.Conv2D(filters=16, kernel_size=5),
11        tf.keras.layers.Activation('sigmoid'),
12        tf.keras.layers.Dropout(0.5, noise_shape=None, seed=None),
13        tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
14        tf.keras.layers.Flatten(),
15        tf.keras.layers.Dense(120),
16        tf.keras.layers.Activation('sigmoid'),
17        tf.keras.layers.Dropout(0.5, noise_shape=None, seed=None),
18        tf.keras.layers.Dense(84),
19        tf.keras.layers.Activation('sigmoid'),
20        tf.keras.layers.Dropout(0.5, noise_shape=None, seed=None),
21        tf.keras.layers.Dense(10, activation = 'softmax'),])
```

```
In [90]: 1 model = net()
2 model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['acc
3 model.fit(x_train,y_train,128,15,validation_split=0.1)
```

Epoch 1/15
422/422 [=====] - 22s 49ms/step - loss: 2.3300 - accur
acy: 0.1034 - val_loss: 2.3018 - val_accuracy: 0.1050
Epoch 2/15
422/422 [=====] - 21s 49ms/step - loss: 2.3028 - accur
acy: 0.1071 - val_loss: 2.3019 - val_accuracy: 0.1050
Epoch 3/15
422/422 [=====] - 20s 48ms/step - loss: 2.3020 - accur
acy: 0.1125 - val_loss: 2.3020 - val_accuracy: 0.1050
Epoch 4/15
422/422 [=====] - 20s 49ms/step - loss: 2.3013 - accur
acy: 0.1129 - val_loss: 2.3013 - val_accuracy: 0.1050
Epoch 5/15
422/422 [=====] - 21s 49ms/step - loss: 1.6285 - accur
acy: 0.4072 - val_loss: 0.4646 - val_accuracy: 0.8658
Epoch 6/15
422/422 [=====] - 21s 49ms/step - loss: 0.6692 - accur
acy: 0.7841 - val_loss: 0.2809 - val_accuracy: 0.9220
Epoch 7/15
422/422 [=====] - 21s 50ms/step - loss: 0.5391 - accur
acy: 0.8336 - val_loss: 0.2260 - val_accuracy: 0.9340
Epoch 8/15
422/422 [=====] - 21s 50ms/step - loss: 0.4713 - accur
acy: 0.8561 - val_loss: 0.1890 - val_accuracy: 0.9447
Epoch 9/15
422/422 [=====] - 21s 50ms/step - loss: 0.4268 - accur
acy: 0.8706 - val_loss: 0.1650 - val_accuracy: 0.9503
Epoch 10/15
422/422 [=====] - 21s 50ms/step - loss: 0.3940 - accur
acy: 0.8790 - val_loss: 0.1453 - val_accuracy: 0.9580
Epoch 11/15
422/422 [=====] - 21s 49ms/step - loss: 0.3631 - accur
acy: 0.8897 - val_loss: 0.1311 - val_accuracy: 0.9607
Epoch 12/15
422/422 [=====] - 21s 49ms/step - loss: 0.3433 - accur
acy: 0.8980 - val_loss: 0.1203 - val_accuracy: 0.9632
Epoch 13/15
422/422 [=====] - 21s 49ms/step - loss: 0.3278 - accur
acy: 0.9018 - val_loss: 0.1116 - val_accuracy: 0.9653
Epoch 14/15
422/422 [=====] - 21s 50ms/step - loss: 0.3114 - accur
acy: 0.9081 - val_loss: 0.1047 - val_accuracy: 0.9687
Epoch 15/15
422/422 [=====] - 21s 50ms/step - loss: 0.2983 - accur
acy: 0.9104 - val_loss: 0.1026 - val_accuracy: 0.9707

Out[90]: <keras.callbacks.History at 0x7f97cc1482d0>

```
In [91]: 1 model.evaluate(x_test,y_test)
```

313/313 [=====] - 2s 6ms/step - loss: 0.1255 - accurac
y: 0.9601

Out[91]: [0.12549354135990143, 0.960099995136261]

The test accuracy of Batch norm model = 0.9772 \ The test accuracy of Standard norm model = 0.9511 \ The test accuracy of Dropout model = 0.9601 \ Therefore the test accuracy of batch norm is greater. And the test accuracy of Standard norm model is the least.

In [92]:

```
1  # Dropout + normalization
2
3  def net():
4      return tf.keras.models.Sequential([
5          tf.keras.layers.BatchNormalization(),
6          tf.keras.layers.Dropout(0.2, noise_shape=None, seed=None),
7          tf.keras.layers.Conv2D(filters=6, kernel_size=5, input_shape=(28, 28,
8          tf.keras.layers.BatchNormalization(),
9          tf.keras.layers.Activation('sigmoid'),
10         tf.keras.layers.Dropout(0.5, noise_shape=None, seed=None),
11         tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
12         tf.keras.layers.Conv2D(filters=16, kernel_size=5),
13         tf.keras.layers.BatchNormalization(),
14         tf.keras.layers.Activation('sigmoid'),
15         tf.keras.layers.Dropout(0.5, noise_shape=None, seed=None),
16         tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
17         tf.keras.layers.Flatten(),
18         tf.keras.layers.Dense(120),
19         tf.keras.layers.BatchNormalization(),
20         tf.keras.layers.Activation('sigmoid'),
21         tf.keras.layers.Dropout(0.5, noise_shape=None, seed=None),
22         tf.keras.layers.Dense(84),
23         tf.keras.layers.BatchNormalization(),
24         tf.keras.layers.Activation('sigmoid'),
25         tf.keras.layers.Dropout(0.5, noise_shape=None, seed=None),
26         tf.keras.layers.Dense(10, activation = 'softmax'),])
```

```
In [93]: 1 model = net()
        2 model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['acc
        3 model.fit(x_train,y_train,128,15,validation_split=0.1)
```

Epoch 1/15
422/422 [=====] - 37s 84ms/step - loss: 1.7867 - accur
acy: 0.3861 - val_loss: 0.7698 - val_accuracy: 0.8473
Epoch 2/15
422/422 [=====] - 35s 83ms/step - loss: 1.0900 - accur
acy: 0.6402 - val_loss: 0.4531 - val_accuracy: 0.8973
Epoch 3/15
422/422 [=====] - 35s 82ms/step - loss: 0.8197 - accur
acy: 0.7383 - val_loss: 0.2889 - val_accuracy: 0.9327
Epoch 4/15
422/422 [=====] - 35s 82ms/step - loss: 0.6366 - accur
acy: 0.8014 - val_loss: 0.2146 - val_accuracy: 0.9437
Epoch 5/15
422/422 [=====] - 35s 83ms/step - loss: 0.5101 - accur
acy: 0.8440 - val_loss: 0.1392 - val_accuracy: 0.9630
Epoch 6/15
422/422 [=====] - 35s 83ms/step - loss: 0.4327 - accur
acy: 0.8706 - val_loss: 0.1141 - val_accuracy: 0.9673
Epoch 7/15
422/422 [=====] - 35s 83ms/step - loss: 0.3844 - accur
acy: 0.8842 - val_loss: 0.0934 - val_accuracy: 0.9730
Epoch 8/15
422/422 [=====] - 35s 83ms/step - loss: 0.3489 - accur
acy: 0.8956 - val_loss: 0.0860 - val_accuracy: 0.9745
Epoch 9/15
422/422 [=====] - 35s 83ms/step - loss: 0.3217 - accur
acy: 0.9046 - val_loss: 0.0727 - val_accuracy: 0.9797
Epoch 10/15
422/422 [=====] - 35s 84ms/step - loss: 0.2991 - accur
acy: 0.9111 - val_loss: 0.0707 - val_accuracy: 0.9788
Epoch 11/15
422/422 [=====] - 35s 83ms/step - loss: 0.2910 - accur
acy: 0.9150 - val_loss: 0.0717 - val_accuracy: 0.9793
Epoch 12/15
422/422 [=====] - 35s 83ms/step - loss: 0.2734 - accur
acy: 0.9186 - val_loss: 0.0642 - val_accuracy: 0.9813
Epoch 13/15
422/422 [=====] - 35s 83ms/step - loss: 0.2653 - accur
acy: 0.9230 - val_loss: 0.0651 - val_accuracy: 0.9782
Epoch 14/15
422/422 [=====] - 35s 83ms/step - loss: 0.2490 - accur
acy: 0.9277 - val_loss: 0.0627 - val_accuracy: 0.9800
Epoch 15/15
422/422 [=====] - 35s 83ms/step - loss: 0.2476 - accur
acy: 0.9276 - val_loss: 0.0561 - val_accuracy: 0.9842

Out[93]: <keras.callbacks.History at 0x7f97cc5e0850>

```
In [94]: 1 model.evaluate(x_test,y_test)
```

313/313 [=====] - 2s 7ms/step - loss: 0.0626 - accurac
y: 0.9795

Out[94]: [0.06260231882333755, 0.9794999957084656]

The test accuracy using both = 0.9795 \ This is greater than the test accuracy achieved using
Dropout alone or Batch normalization alone.

```
In [ ]: 1
```

Problem 4

```
In [102]: 1 import time
2 import torch
3 import matplotlib.pyplot as plt
4 from time import time
5 from torch import nn, optim
6 import torch.nn.functional as F
7 from sklearn.model_selection import train_test_split
8 from torch.utils.data import DataLoader, TensorDataset
9 import time
```

```
In [96]: 1 def f(x1,x2):
2     a=np.sqrt(np.fabs(x2+x1/2+47))
3     b=np.sqrt(np.fabs(x1-(x2+47)))
4     c=-(x2+47)*np.sin(a)-x1*np.sin(b)
5     return c
```

```
In [97]: 1 def count_parameters(model):
2     return sum(p.numel() for p in model.parameters() if p.requires_grad)
3 # count_parameters(model3)
```

```
In [98]: 1 def train(model,dataloader,epochs):
2     for ep in range(epochs):
3         ep_loss = 0
4         for x_b,y_b in dataloader:
5             optimizer.zero_grad()
6             # Forward pass
7             y_pred = model(x_b)
8             # Compute Loss
9             # print(y_pred,y_train)
10            loss = criterion(y_pred.squeeze(), y_b)
11            # loss = torch.sqrt(loss)
12            ep_loss += loss/BATCH_SIZE
13            # Backward pass
14            loss.backward()
15            optimizer.step()
16            # print('Epoch {}: train Loss: {}'.format(ep, ep_loss))
17            # print(epoch)
18 def test(model,x_test,y_test):
19     loss=0
20     y_pred = model((x_test))
21     loss = criterion(y_pred.squeeze(),y_test)
22     return (torch.sqrt(loss))
```

```
In [133]: 1 n = 100000
2 x1 = np.random.uniform(-512,512,n)
3 x2 = np.random.uniform(-512,512,n)
4 x = torch.FloatTensor([([x1[i],x2[i]]) for i in range(n)])
5 # print(x)
6 y = torch.FloatTensor(f(x1,x2) + np.random.normal(0,np.sqrt(0.3),n))
7 x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.2)
8 dataset = TensorDataset(x_train, y_train)
9 BATCH_SIZE=1000
10 dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

```
In [64]: 1 class Feedforward_1layer(torch.nn.Module):
2     def __init__(self, input_size, hidden_size):
3         super(Feedforward_1layer, self).__init__()
4         self.input_size = input_size
5         self.hidden_size = hidden_size
6         self.fc1 = torch.nn.Linear(self.input_size, self.hidden_size)
7         self.relu = torch.nn.ReLU()
8         self.fc2 = torch.nn.Linear(self.hidden_size, 1)
9         self.sigmoid = torch.nn.Sigmoid()
10    def forward(self, x):
11        hidden = self.fc1(x)
12        relu = self.relu(hidden)
13        output = self.fc2(relu)
14        return output
```

```
In [65]: 1 class Feedforward_2layer(torch.nn.Module):
2         def __init__(self, input_size, hidden_size1, hidden_size2):
3             super(Feedforward_2layer, self).__init__()
4             self.input_size = input_size
5             self.hidden_size1 = hidden_size1
6             self.hidden_size2 = hidden_size2
7             self.fc1 = torch.nn.Linear(self.input_size, self.hidden_size1)
8             self.relu = torch.nn.ReLU()
9             self.fc2 = torch.nn.Linear(self.hidden_size1, self.hidden_size2)
10            self.fc3 = torch.nn.Linear(self.hidden_size2, 1)
11            self.sigmoid = torch.nn.Sigmoid()
12            self.batch_normalization1 = torch.nn.BatchNorm1d(self.hidden_size1)
13            self.batch_normalization2 = torch.nn.BatchNorm1d(self.hidden_size2)
14
15            def forward(self, x):
16                hidden1 = self.fc1(x)
17                hidden1 = self.batch_normalization1(hidden1)
18                relu = self.relu(hidden1)
19                hidden2 = self.fc2(relu)
20                hidden2 = self.batch_normalization2(hidden2)
21                relu = self.relu(hidden2)
22                output = self.fc3(relu)
23                return output
```

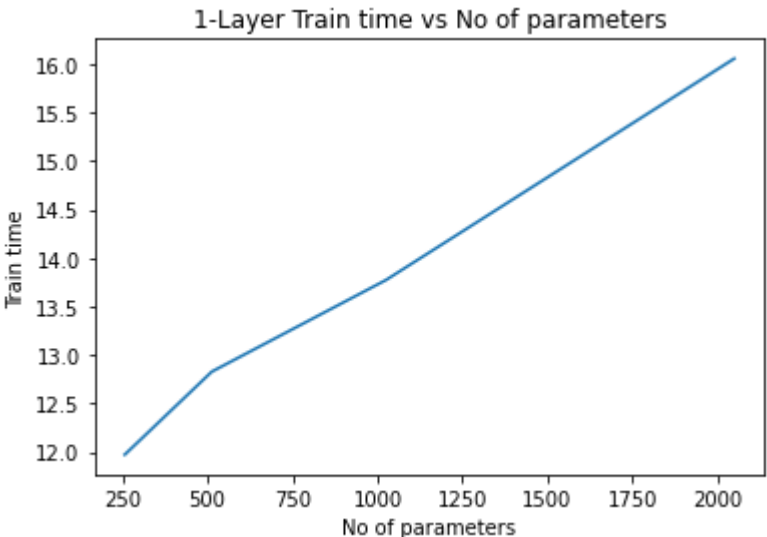
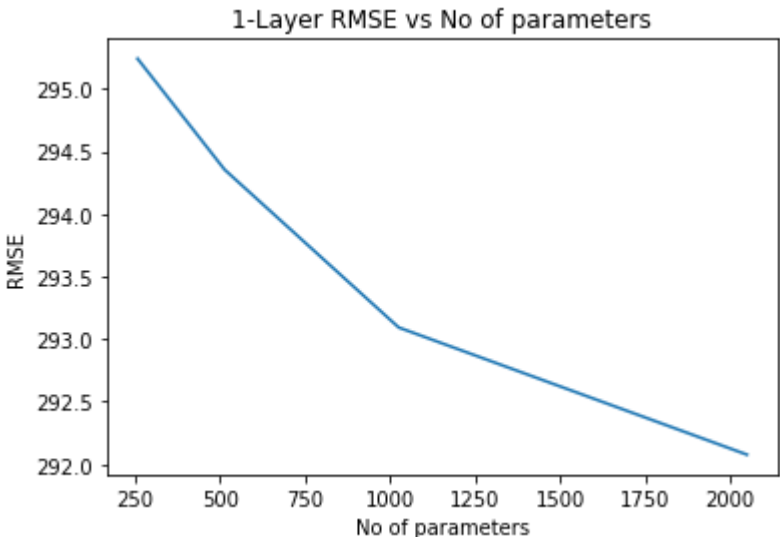
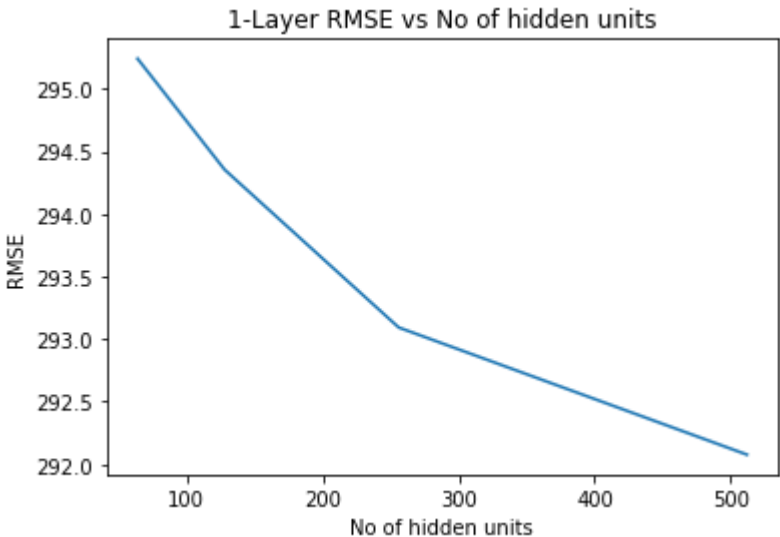
```
In [66]: 1 class Feedforward_3layer(torch.nn.Module):
2         def __init__(self, input_size, hidden_size1, hidden_size2, hidden_size3):
3             super(Feedforward_3layer, self).__init__()
4             self.input_size = input_size
5             self.hidden_size1 = hidden_size1
6             self.hidden_size2 = hidden_size2
7             self.hidden_size3 = hidden_size3
8             self.fc1 = torch.nn.Linear(self.input_size, self.hidden_size1)
9             self.relu = torch.nn.ReLU()
10            self.fc2 = torch.nn.Linear(self.hidden_size1, self.hidden_size2)
11            self.fc3 = torch.nn.Linear(self.hidden_size2, self.hidden_size3)
12            self.fc4 = torch.nn.Linear(self.hidden_size3, 1)
13            self.sigmoid = torch.nn.Sigmoid()
14            self.batch_normalization1 = torch.nn.BatchNorm1d(self.hidden_size1)
15            self.batch_normalization2 = torch.nn.BatchNorm1d(self.hidden_size2)
16            self.batch_normalization3 = torch.nn.BatchNorm1d(self.hidden_size3)
17
18            def forward(self, x):
19                hidden1 = self.fc1(x)
20                hidden1 = self.batch_normalization1(hidden1)
21                relu = self.relu(hidden1)
22                hidden2 = self.fc2(relu)
23                hidden2 = self.batch_normalization2(hidden2)
24                relu = self.relu(hidden2)
25                hidden3 = self.fc3(relu)
26                hidden3 = self.batch_normalization3(hidden3)
27                relu = self.relu(hidden3)
28                output = self.fc4(relu)
29                return output
```

```
In [103]: 1
2         criterion = torch.nn.MSELoss()
3
```

```
In [12]: 1 hiddensize_1 = [64,128,256,512]
2 no_params_1 = []
3 loss_1 = []
4 time_1 = []
5 for h in hiddensize_1:
6     model = Feedforward_1layer(2,h)
7     optimizer = torch.optim.SGD(model.parameters(),lr=1e-7,momentum=0.9,damp
8     start = time.time()
9     train(model,dataloader,50)
10    end = time.time()
11    time_1.append(end-start)
12    no_params_1.append(count_parameters(model))
13    loss_1.append(test(model,x_test,y_test))
```


In [34]:

```
1 plt.plot(hiddensize_1,loss_1)
2 plt.title('1-Layer RMSE vs No of hidden units')
3 plt.xlabel('No of hidden units')
4 plt.ylabel('RMSE')
5 plt.show()
6 plt.plot(no_params_1,loss_1)
7 plt.title('1-Layer RMSE vs No of parameters')
8 plt.xlabel('No of parameters')
9 plt.ylabel('RMSE')
10 plt.show()
11 plt.plot(no_params_1,time_1)
12 plt.title('1-Layer Train time vs No of parameters')
13 plt.xlabel('No of parameters')
14 plt.ylabel('Train time')
15 plt.show()
```



In [184]:

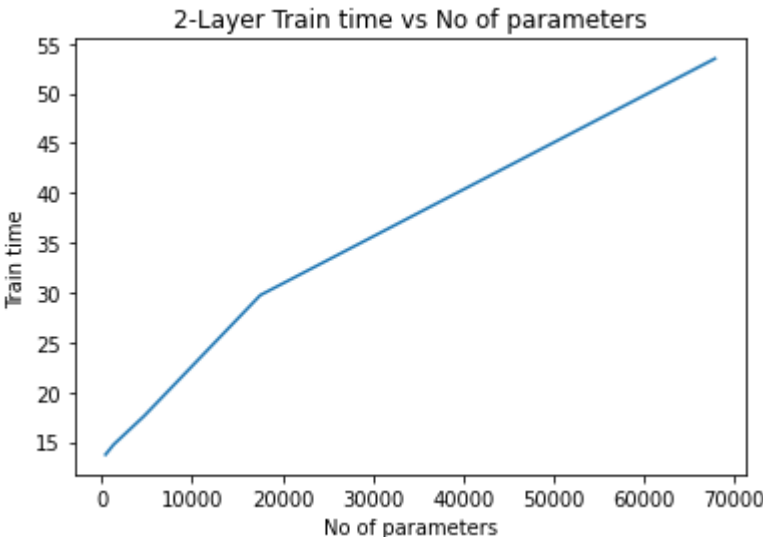
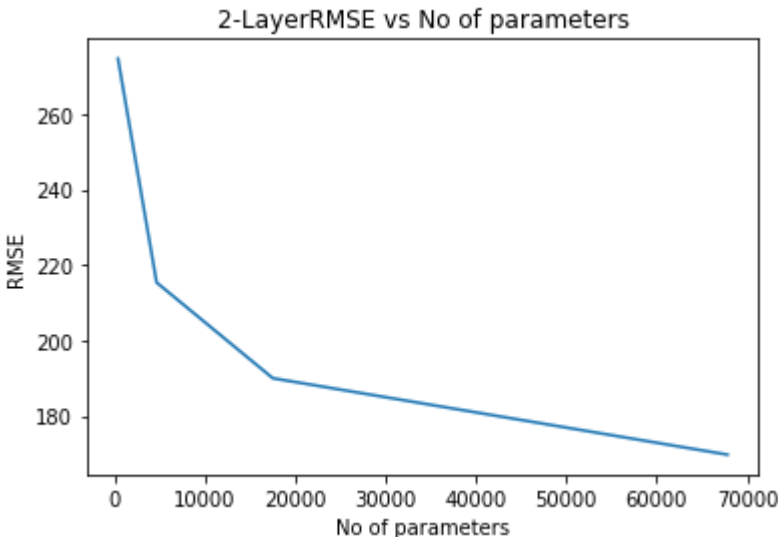
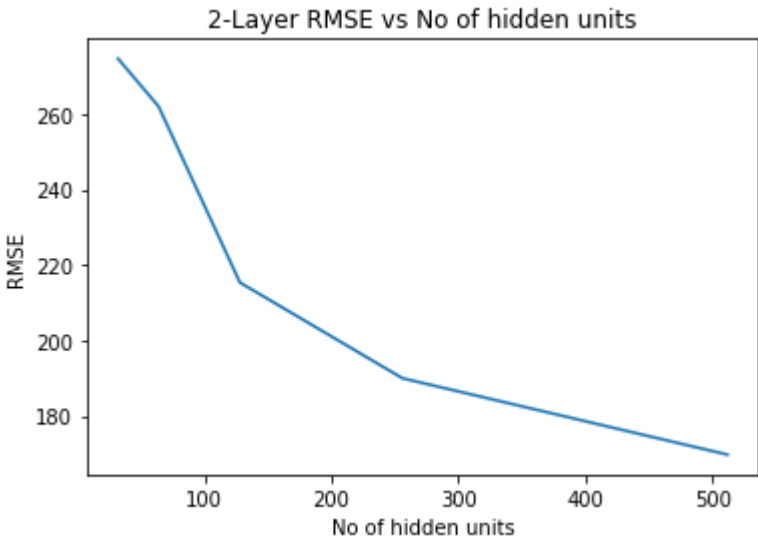
```
1
```

In [107]:

```
1 hiddensize_2 = [[16,16],[32,32],[64,64],[128,128],[256,256]]
2 no_params_2 = []
3 loss_2 = []
4 time_2 = []
5 for h in hiddensize_2:
6     model = Feedforward_2layer(2,h[0],h[1])
7     optimizer = torch.optim.SGD(model.parameters(),lr=1e-4,momentum=0.9,damp
8     start = time.time()
9     train(model,dataloader,20)
10    end = time.time()
11    time_2.append(end-start)
12    no_params_2.append(count_parameters(model))
13    loss_2.append(test(model,x_test,y_test))
```

In [108]:

```
1 plt.plot([sum(hiddensize_2[i]) for i in range(len(hiddensize_2))],loss_2)
2 plt.title('2-Layer RMSE vs No of hidden units')
3 plt.xlabel('No of hidden units')
4 plt.ylabel('RMSE')
5 plt.show()
6 plt.plot(no_params_2,loss_2)
7 plt.title('2-LayerRMSE vs No of parameters')
8 plt.xlabel('No of parameters')
9 plt.ylabel('RMSE')
10 plt.show()
11 plt.plot(no_params_2,time_2)
12 plt.title('2-Layer Train time vs No of parameters')
13 plt.xlabel('No of parameters')
14 plt.ylabel('Train time')
15 plt.show()
```

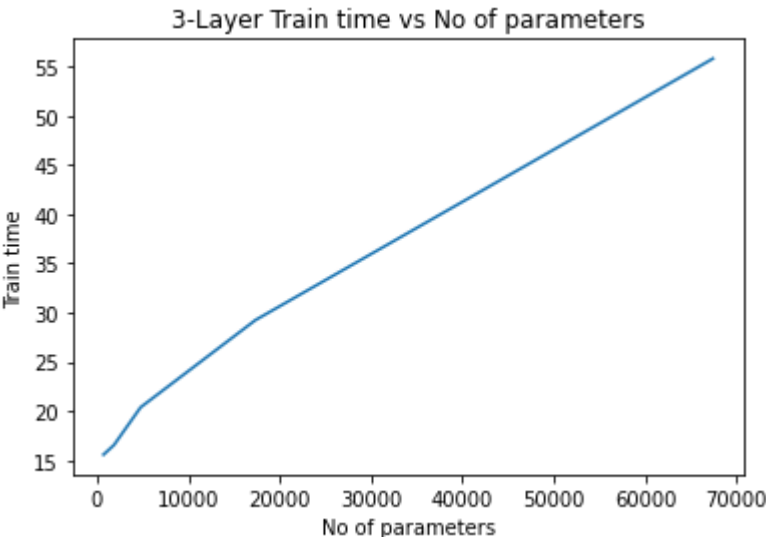
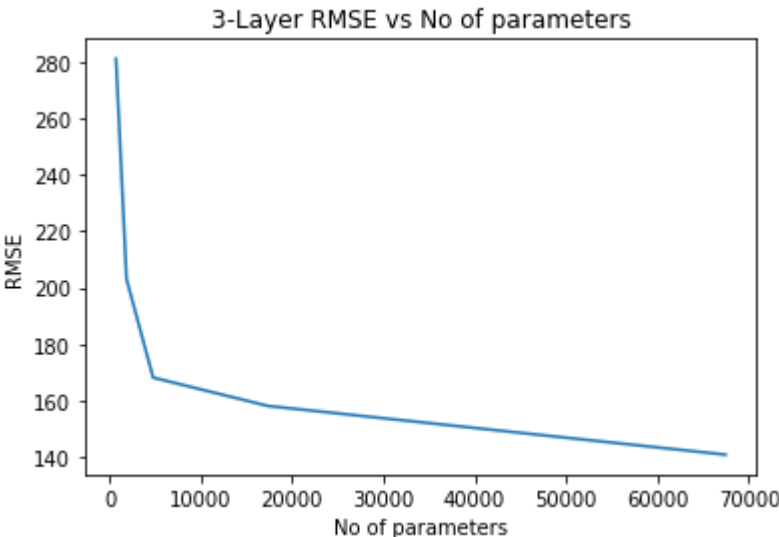
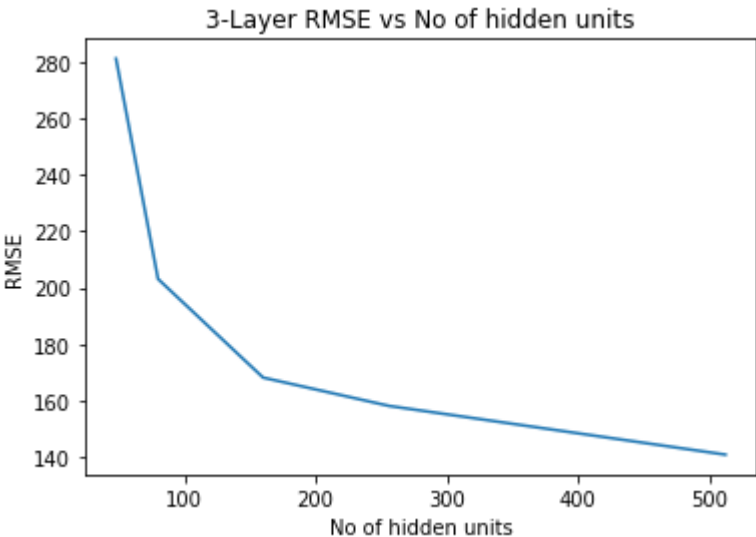


In [186]:

```
1
```

```
In [140]: 1 hiddensize_3 = [[16,16,16],[32,32,16],[64,32,64],[64,128,64],[128,256,128]]
2 no_params_3 = []
3 loss_3 = []
4 time_3 = []
5 for h in hiddensize_3:
6     model = Feedforward_3layer(2,h[0],h[1],h[2])
7     optimizer = torch.optim.SGD(model.parameters(),lr=1e-4,momentum=0.9,damp
8     start = time.time()
9     train(model,dataloader,20)
10    end = time.time()
11    time_3.append(end-start)
12    no_params_3.append(count_parameters(model))
13    loss_3.append(test(model,x_test,y_test))
```

```
In [141]: 1 plt.plot([sum(hiddensize_3[i]) for i in range(len(hiddensize_3))],loss_3)
2 plt.title('3-Layer RMSE vs No of hidden units')
3 plt.xlabel('No of hidden units')
4 plt.ylabel('RMSE')
5 plt.show()
6 plt.plot(no_params_3,loss_3)
7 plt.title('3-Layer RMSE vs No of parameters')
8 plt.xlabel('No of parameters')
9 plt.ylabel('RMSE')
10 plt.show()
11 plt.plot(no_params_3,time_3)
12 plt.title('3-Layer Train time vs No of parameters')
13 plt.xlabel('No of parameters')
14 plt.ylabel('Train time')
15 plt.show()
```



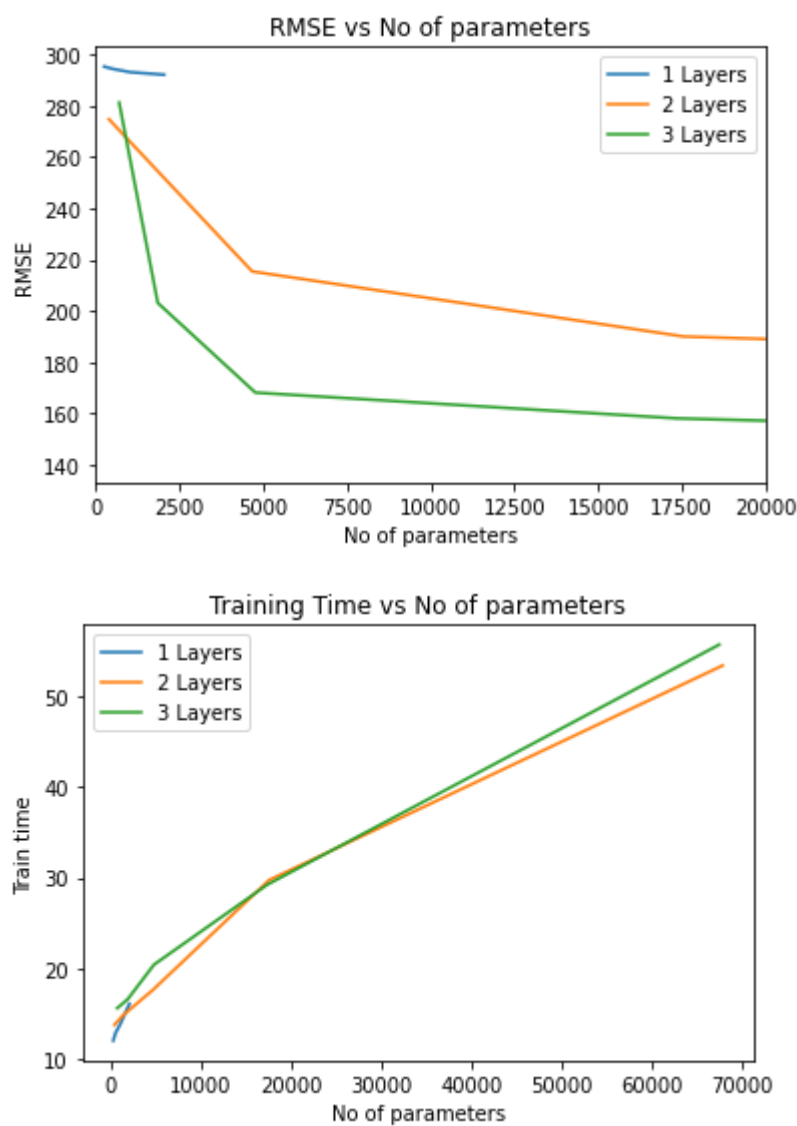
```
In [124]: 1
```

As we go from deeper to shallow networks as the number of parameters increase , the amount of decrease in the RMSE is lesser. This can be clearly observed in the graph below. With a small increase in the number of parameters in the 3 layer model there is high decrease in the RMSE

when compared to that of 2 layer and 1 layer models. And comparatively 2 layer model has a higher decrease in RMSE wen compared to 1 layer model. \ The training time increases as the number of layers increases. We can observe that the slope of the 1 layer model is very high when compared to 2 and 3 layer models. But the training time is higher for 2 layer model when compared to 1 layer model. And the train time of the 2 layer model is very close but slightly lesser than that of the 3 layer model.

In [145]:

```
1 plt.plot(no_params_1,loss_1,label='1 Layers')
2 plt.plot(no_params_2,loss_2,label='2 Layers')
3 plt.plot(no_params_3,loss_3,label='3 Layers')
4 plt.title('RMSE vs No of parameters')
5 plt.xlabel('No of parameters')
6 plt.ylabel('RMSE')
7 plt.xlim(0,20000)
8 plt.legend()
9 plt.show()
10
11 plt.plot(no_params_1,time_1,label='1 Layers')
12 plt.plot(no_params_2,time_2,label='2 Layers')
13 plt.plot(no_params_3,time_3,label='3 Layers')
14 plt.title('Training Time vs No of parameters')
15 plt.xlabel('No of parameters')
16 plt.ylabel('Train time')
17 # plt.xlim(0,10000)
18 plt.legend()
19 plt.show()
```



In []:

```
1
```