Multiple MLP Architectures on MNIST

December 8, 2018

1 Multiple MLP Architectures using Keras on MNIST

1.1 Purpose

The purpose of the study is to try out **3 different MLP architectures on MNIST dataset to compare the performance.** The implementation is done in Keras.

1.2 Steps at a Glance:

1. Take the famous MNIST dataset as input. http://yann.lecun.com/exdb/mnist/
2. Feed it into **2-layered MLP Architecture: Input(784)-ReLu(512)-ReLu(128)-Sigmoid(output)**
3. Find the accuracy and draw the **Loss vs Epoch Plot**
4. Introduce **Batch Normalization and Dropouts.**
5. Evaluate the model again by estimating accuracy and drawing loss diagram.
6. Feed same input to **3 layered MLP Architecture: Input(784)-ReLu(512)-ReLu(256)-ReLu(64)-Sigmoid(output)**
7. Introduce Batch Normalization and Dropouts & evaluate the model again.
8. Feed same input to **5 layered MLP Architecture: Input(784)-ReLu(512)-ReLu(256)-ReLu(144)-ReLu(96)-ReLu(36)-Sigmoid(output)**
9. Introduce Batch Normalization and Dropouts & evaluate the model again.
10. Analyze the output from the above 3 architectures and draw conclusions .

In [53]:
```
# if you keras is not using tensorflow as backend
# set "KERAS_BACKEND=tensorflow" use this command
from keras.utils import np_utils
from keras.datasets import mnist
import seaborn as sns
from keras.initializers import RandomNormal
```

1.3 Loading and Pre-processing Data

In [54]:
```
# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

In [55]:
```
print("Number of training examples :", X_train.shape[0],
      "and each image is of shape (%d, %d)"%(X_train.shape[1], X_train.shape[2]))
print("Number of testing examples :", X_test.shape[0],
      "and each image is of shape (%d, %d)"%(X_test.shape[1], X_test.shape[2]))
```

```
Number of training examples : 60000 and each image is of shape (28, 28)
Number of testing examples : 10000 and each image is of shape (28, 28)
```

In [56]:
```python
# if you observe the input shape its 3 dimensional vector
# for each image we have a (28*28) vector
# we will convert the (28*28) vector into single dimensional vector of 1 * 784

X_train = X_train.reshape(X_train.shape[0], X_train.shape[1]*X_train.shape[2])
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1]*X_test.shape[2])
```

In [57]:
```python
# after converting the input images from 3d to 2d vectors

print("Number of training examples :", X_train.shape[0],
         "and each image is of shape (%d)"%(X_train.shape[1]))
print("Number of testing examples :", X_test.shape[0],
         "and each image is of shape (%d)"%(X_test.shape[1]))
```

```
Number of training examples : 60000 and each image is of shape (784)
Number of testing examples : 10000 and each image is of shape (784)
```

In [58]:
```python
# An example data point
print(X_train[0])
```

```
[   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   3  18  18  18 126 136 175  26 166 255
  247 127   0   0   0   0   0   0   0   0   0   0   0   0  30  36  94 154
  170 253 253 253 253 253 225 172 253 242 195  64   0   0   0   0   0   0
    0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93  82
   82  56  39   0   0   0   0   0   0   0   0   0   0   0   0  18 219 253
  253 253 253 253 198 182 247 241   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0  14   1 154 253  90   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0  11 190 253  70   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  35 241
  225 160 108   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0  81 240 253 253 119  25   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
   0   0  45 186 253 253 150  27   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253 187
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0 249 253 249  64   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253
 253 207   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0  39 148 229 253 253 253 250 182   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253
 253 201  78   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0  23  66 213 253 253 253 253 198  81   2   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0  18 171 219 253 253 253 253 195
  80   9   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  55 172 226 253 253 253 253 244 133  11   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0 136 253 253 253 212 135 132  16
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0]
```

In [59]:
```python
# if we observe the above matrix each cell is having a value between 0-255
# before we move to apply machine learning algorithms lets try to normalize the data
# X => (X - Xmin)/(Xmax-Xmin) = X/255

X_train = X_train/255
X_test = X_test/255
```

In [60]:
```python
# example data point after normlizing
print(X_train[0])
```

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
```

```
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.01176471 0.07058824 0.07058824 0.07058824
0.49411765 0.53333333 0.68627451 0.10196078 0.65098039 1.
0.96862745 0.49803922 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.11764706 0.14117647 0.36862745 0.60392157
0.66666667 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
0.88235294 0.6745098  0.99215686 0.94901961 0.76470588 0.25098039
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.19215686
0.93333333 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
0.99215686 0.99215686 0.99215686 0.98431373 0.36470588 0.32156863
0.32156863 0.21960784 0.15294118 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.07058824 0.85882353 0.99215686
0.99215686 0.99215686 0.99215686 0.99215686 0.77647059 0.71372549
0.96862745 0.94509804 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.31372549 0.61176471 0.41960784 0.99215686
0.99215686 0.80392157 0.04313725 0.         0.16862745 0.60392157
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.05490196 0.00392157 0.60392157 0.99215686 0.35294118
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.54509804 0.99215686 0.74509804 0.00784314 0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.04313725
0.74509804 0.99215686 0.2745098  0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.1372549  0.94509804
0.88235294 0.62745098 0.42352941 0.00392157 0.         0.
```

```
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.31764706 0.94117647 0.99215686
0.99215686 0.46666667 0.09803922 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.17647059 0.72941176 0.99215686 0.99215686
0.58823529 0.10588235 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.0627451  0.36470588 0.98823529 0.99215686 0.73333333
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.97647059 0.99215686 0.97647059 0.25098039 0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.18039216 0.50980392 0.71764706 0.99215686
0.99215686 0.81176471 0.00784314 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.15294118 0.58039216
0.89803922 0.99215686 0.99215686 0.99215686 0.98039216 0.71372549
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.09411765 0.44705882 0.86666667 0.99215686 0.99215686 0.99215686
0.99215686 0.78823529 0.30588235 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.09019608 0.25882353 0.83529412 0.99215686
0.99215686 0.99215686 0.99215686 0.77647059 0.31764706 0.00784314
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.07058824 0.67058824
0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.76470588
0.31372549 0.03529412 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.21568627 0.6745098  0.88627451 0.99215686 0.99215686 0.99215686
0.99215686 0.95686275 0.52156863 0.04313725 0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
```

```
 0.         0.         0.         0.         0.53333333 0.99215686
 0.99215686 0.99215686 0.83137255 0.52941176 0.51764706 0.0627451
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]
```

In [61]:
```
# here we are having a class number for each image
print("Class label of first image :", y_train[0])

# lets convert this into a 10 dimensional vector
# ex: consider an image is 5 convert it into 5 => [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
# this conversion needed for MLPs

Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)

print("After converting the output into a vector : ",Y_train[0])
```

```
Class label of first image : 5
After converting the output into a vector :  [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

In [62]:
```
# https://keras.io/getting-started/sequential-model-guide/

# The Sequential model is a linear stack of layers.
# you can create a Sequential model by passing
# a list of layer instances to the constructor:

# model = Sequential([
#     Dense(32, input_shape=(784,)),
#     Activation('relu'),
#     Dense(10),
#     Activation('softmax'),
```

```python
# ])

# You can also simply add layers via the .add() method:

# model = Sequential()
# model.add(Dense(32, input_dim=784))
# model.add(Activation('relu'))

###

# https://keras.io/layers/core/

# keras.layers.Dense(units, activation=None, use_bias=True,
# kernel_initializer='glorot_uniform', bias_initializer='zeros',
# kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
# kernel_constraint=None, bias_constraint=None)

# Dense implements the operation: output = activation(dot(input, kernel) + bias)
# where activation is the element-wise activation function passed as the activation
# argument, kernel is a weights matrix created by the layer, and
# bias is a bias vector created by the layer (only applicable if use_bias is True).

# output = activation(dot(input, kernel) + bias)  => y = activation(WT. X + b)

####

# https://keras.io/activations/

# Activations can either be used through an Activation layer, or through
# the activation argument supported by all forward layers:

# from keras.layers import Activation, Dense

# model.add(Dense(64))
# model.add(Activation('tanh'))

# This is equivalent to:
# model.add(Dense(64, activation='tanh'))

# there are many activation functions  available ex: tanh, relu, softmax


from keras.models import Sequential
from keras.layers import Dense, Activation
```

2 Initializations

```
In [63]: # initialization of some model parameters

         output_dim = 10
         input_dim = X_train.shape[1]

         batch_size = 128
         nb_epoch = 20
```

3 Custom-Defined Functions

```
In [64]: %matplotlib inline
         import matplotlib.pyplot as plt
         import numpy as np
         import time
         # https://gist.github.com/greydanus/f6eee59eaf1d90fcb3b534a25362cea4
         # https://stackoverflow.com/a/14434334
         # this function is used to update the plots for each epoch and error
         def plt_dynamic(fig, x, vy, ty, ax, colors=['b']):
             ax.plot(x, vy, 'b', label="Validation Loss")
             ax.plot(x, ty, 'r', label="Train Loss")
             plt.legend()
             plt.grid()
             fig.canvas.draw()
```

```
In [65]: # To train the model using Adam
         # This function is common to all models.
         def trainModel(model):
             model.compile(optimizer='adam',
                           loss='categorical_crossentropy', metrics=['accuracy'])

             history = model.fit(X_train, Y_train, batch_size=batch_size,
                         epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

             return history
```

```
In [66]: # To plot the Train & Test loss graph.
         # This function is common to all models.
         def plotGraph(model, history):
             score = model.evaluate(X_test, Y_test, verbose=0)
             print('Test score:', score[0])
             print('Test accuracy:', score[1])

             fig,ax = plt.subplots(1,1)
             ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')

             # list of epoch numbers
```

```
x = list(range(1,nb_epoch+1))

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size,
        #epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

# we will get val_loss and val_acc only
# when you pass the paramter validation_data
# val_loss : validation loss
# val_acc : validation accuracy

# loss : training loss
# acc : train accuracy
# for each key in histrory.histrory we will have
# a list of length equal to number of epochs

vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(fig, x, vy, ty, ax)
```

3.1 Model 1: (2-layered MLP Architecture)

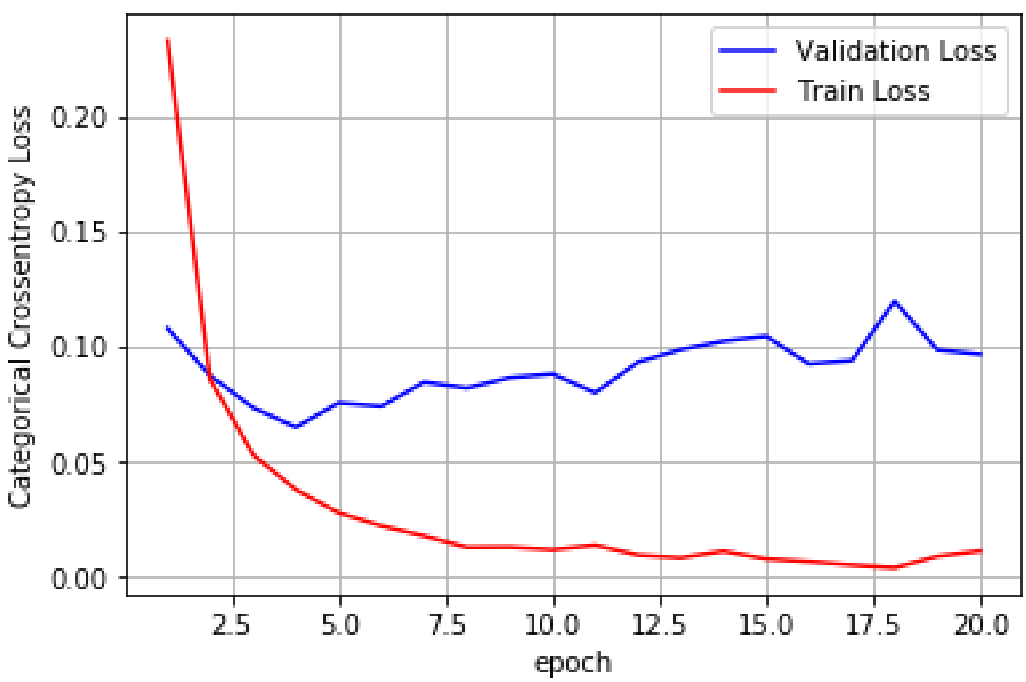
```
In [67]: def plotWeightM1(model):
             w_after = model.get_weights()

             h1_w = w_after[0].flatten().reshape(-1,1)
             h2_w = w_after[2].flatten().reshape(-1,1)
             out_w = w_after[4].flatten().reshape(-1,1)


             fig = plt.figure()
             plt.title("Weight matrices after model trained")
             plt.subplot(1, 3, 1)
             plt.title("Trained Weights")
             ax = sns.violinplot(y=h1_w,color='b')
             plt.xlabel('Hidden Layer 1')

             plt.subplot(1, 3, 2)
             plt.title("Trained Weights")
             ax = sns.violinplot(y=h2_w, color='r')
             plt.xlabel('Hidden Layer 2 ')

             plt.subplot(1, 3, 3)
             plt.title("Trained Weights")
             ax = sns.violinplot(y=out_w,color='y')
             plt.xlabel('Output Layer ')
             plt.show()
```

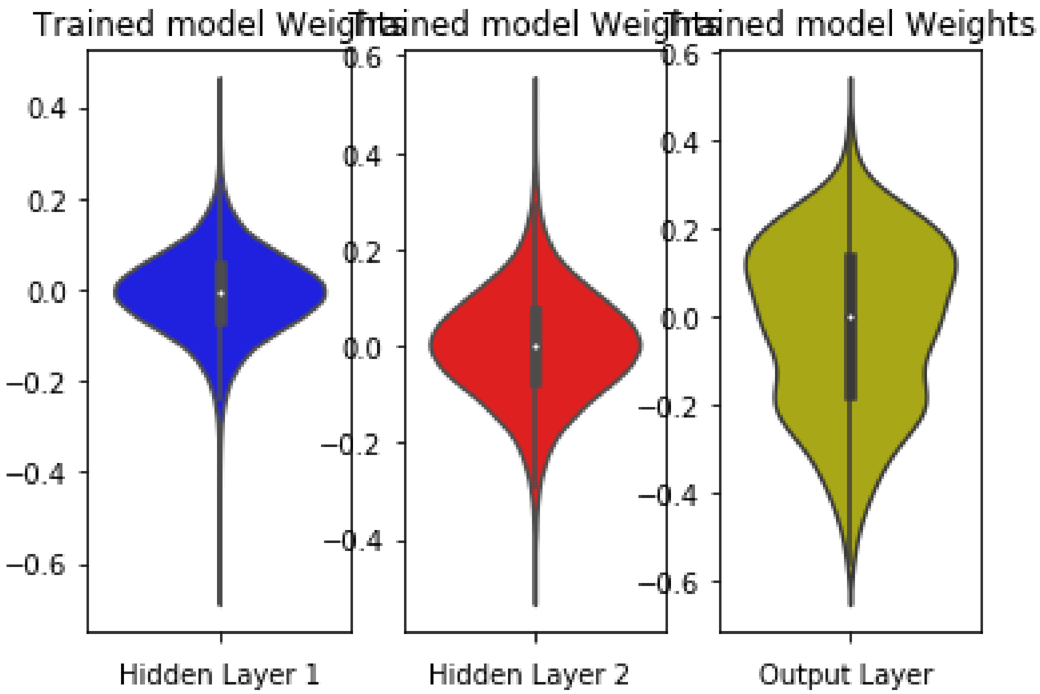
3.1.1 Input(784)-ReLu(512)-ReLu(128)-Softmax(output)

```
In [68]: model_relu = Sequential()
         model_relu.add(Dense(512, activation='relu',
                              input_shape=(input_dim,), kernel_initializer='he_normal'))
         model_relu.add(Dense(128, activation='relu', kernel_initializer='he_normal'))
         model_relu.add(Dense(output_dim, activation='softmax'))

         print(model_relu.summary())

         history = trainModel(model=model_relu)
         plotGraph(model=model_relu, history=history)
         plotWeightM1(model=model_relu)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_79 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_80 (Dense)             (None, 128)               65664
_________________________________________________________________
dense_81 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 16s 268us/step - loss: 0.2330 - acc: 0.9315 - va
Epoch 2/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0859 - acc: 0.9741 - val_
Epoch 3/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0531 - acc: 0.9836 - val_
Epoch 4/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0380 - acc: 0.9883 - val_
Epoch 5/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0280 - acc: 0.9910 - val_
Epoch 6/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0223 - acc: 0.9929 - val_
Epoch 7/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0179 - acc: 0.9942 - val_
Epoch 8/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.0130 - acc: 0.9959 - val_
Epoch 9/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0131 - acc: 0.9956 - val_
Epoch 10/20
60000/60000 [==============================] - 4s 58us/step - loss: 0.0120 - acc: 0.9963 - val_
```

```
Epoch 11/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0139 - acc: 0.9956 - val_
Epoch 12/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0096 - acc: 0.9966 - val_
Epoch 13/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0085 - acc: 0.9972 - val_
Epoch 14/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0113 - acc: 0.9964 - val_
Epoch 15/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0079 - acc: 0.9973 - val_
Epoch 16/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0068 - acc: 0.9979 - val_
Epoch 17/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0053 - acc: 0.9986 - val_
Epoch 18/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0041 - acc: 0.9988 - val_
Epoch 19/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0091 - acc: 0.9971 - val_
Epoch 20/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0114 - acc: 0.9964 - val_
Test score: 0.096843903848933
Test accuracy: 0.9812
```





3.1.2 Model 1: M1 + Batch-Normalization on hidden Layers

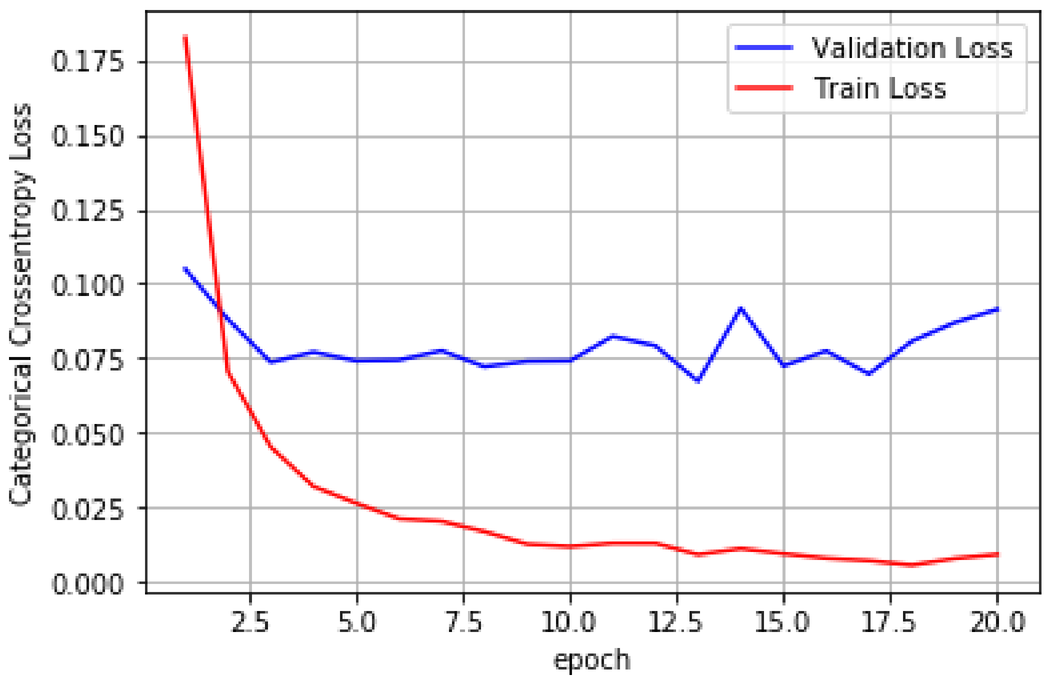
In [69]:
```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,)
# we satisfy this condition with =(2/(ni+ni+1).
# h1 =>  =(2/(ni+ni+1) = 0.039  => N(0,) = N(0,0.039)
# h2 =>  =(2/(ni+ni+1) = 0.055  => N(0,) = N(0,0.055)
# h1 =>  =(2/(ni+ni+1) = 0.120  => N(0,) = N(0,0.120)

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(512, activation='relu',
                    input_shape=(input_dim,), kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(128, activation='relu', kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(output_dim, activation='softmax'))
```

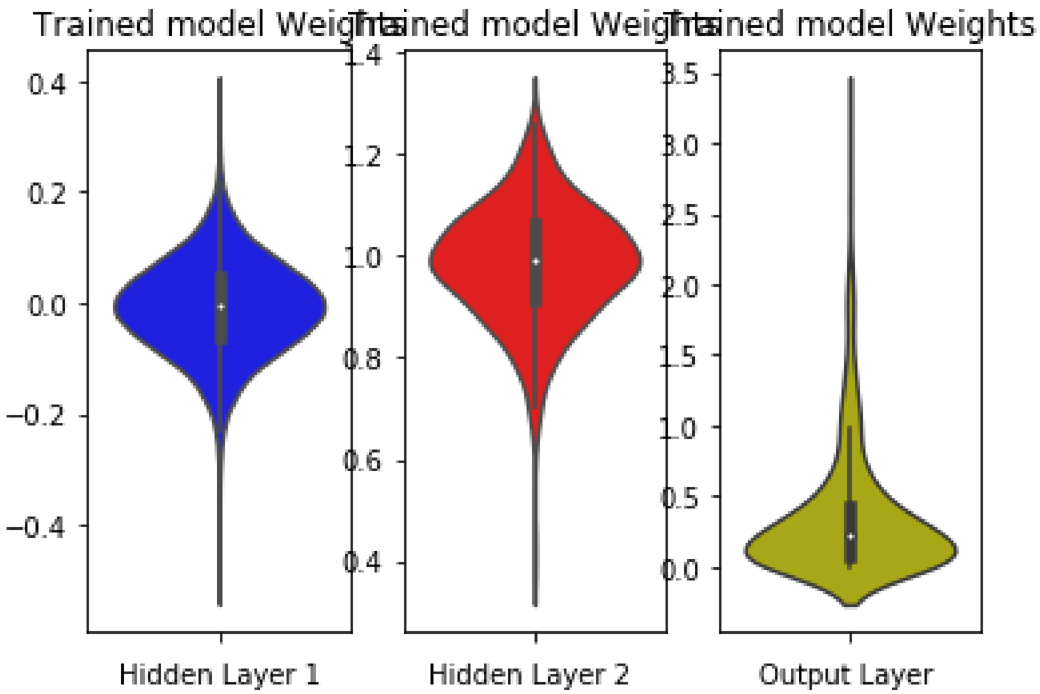
```
model_batch.summary()
```

```
history = trainModel(model=model_batch)
plotGraph(model=model_batch, history=history)
plotWeightM1(model=model_batch)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_82 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_41 (Batc (None, 512)               2048
_________________________________________________________________
dense_83 (Dense)             (None, 128)               65664
_________________________________________________________________
batch_normalization_42 (Batc (None, 128)               512
_________________________________________________________________
dense_84 (Dense)             (None, 10)                1290
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 17s 276us/step - loss: 0.1825 - acc: 0.9455 - va
Epoch 2/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0705 - acc: 0.9786 - val_
Epoch 3/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0453 - acc: 0.9863 - val_
Epoch 4/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0321 - acc: 0.9903 - val_
Epoch 5/20
60000/60000 [==============================] - 4s 75us/step - loss: 0.0265 - acc: 0.9918 - val_
Epoch 6/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0213 - acc: 0.9930 - val_
Epoch 7/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0204 - acc: 0.9933 - val_
Epoch 8/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0170 - acc: 0.9948 - val_
Epoch 9/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0128 - acc: 0.9959 - val_
Epoch 10/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0119 - acc: 0.9964 - val_
Epoch 11/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0131 - acc: 0.9958 - val_
Epoch 12/20
```

```
60000/60000 [==============================] - 4s 66us/step - loss: 0.0131 - acc: 0.9960 - val_
Epoch 13/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0093 - acc: 0.9970 - val_
Epoch 14/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0113 - acc: 0.9965 - val_
Epoch 15/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0095 - acc: 0.9966 - val_
Epoch 16/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0080 - acc: 0.9974 - val_
Epoch 17/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0072 - acc: 0.9976 - val_
Epoch 18/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0057 - acc: 0.9982 - val_
Epoch 19/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0079 - acc: 0.9973 - val_
Epoch 20/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0093 - acc: 0.9969 - val_
Test score: 0.09142072524498654
Test accuracy: 0.9795
```



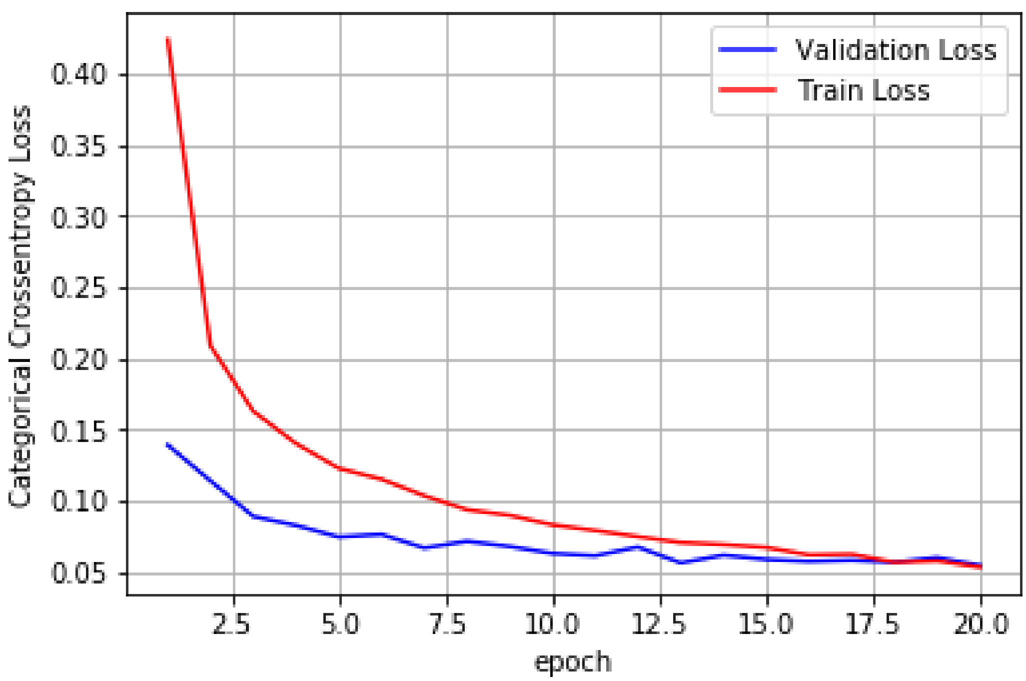


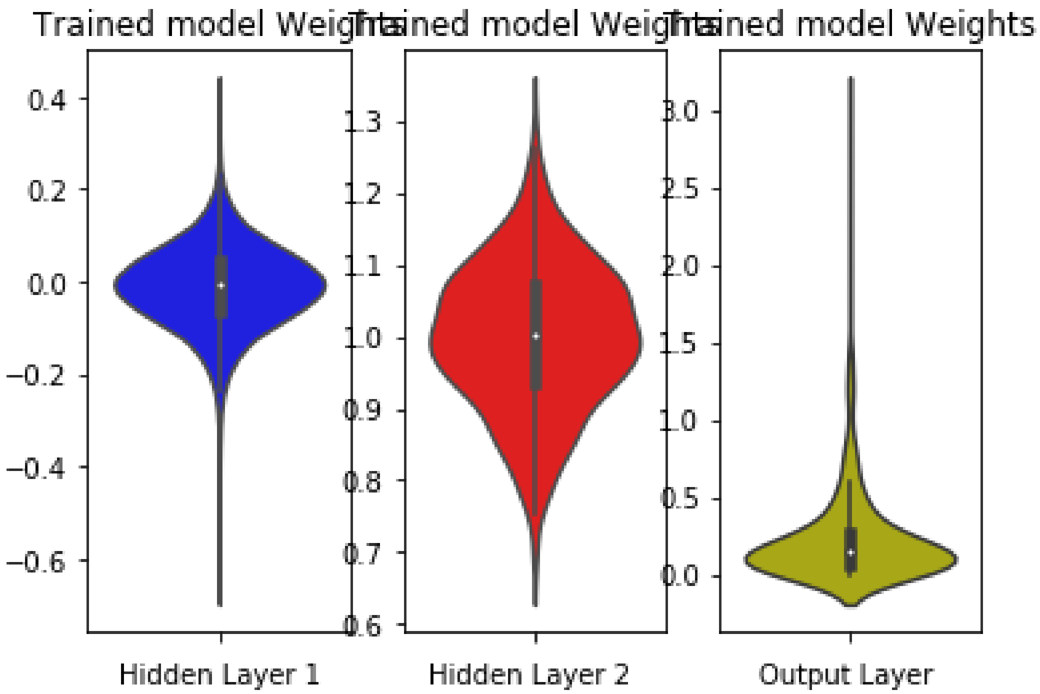
3.1.3 Model 1: M1 + Batch-Normalization + Dropout

In [70]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='relu',
                     input_shape=(input_dim,), kernel_initializer='he_normal'))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer='he_normal'))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(output_dim, activation='softmax'))

model_drop.summary()

history = trainModel(model=model_drop)
plotGraph(model=model_drop, history=history)
plotWeightM1(model=model_drop)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_85 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_43 (Batc (None, 512)               2048
_________________________________________________________________
dropout_21 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_86 (Dense)             (None, 128)               65664
_________________________________________________________________
batch_normalization_44 (Batc (None, 128)               512
_________________________________________________________________
dropout_22 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_87 (Dense)             (None, 10)                1290
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 17s 281us/step - loss: 0.4242 - acc: 0.8725 - va
Epoch 2/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.2088 - acc: 0.9374 - val_
Epoch 3/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.1627 - acc: 0.9511 - val_
Epoch 4/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.1401 - acc: 0.9576 - val_
Epoch 5/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.1227 - acc: 0.9627 - val_
Epoch 6/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.1150 - acc: 0.9650 - val_
Epoch 7/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.1033 - acc: 0.9684 - val_
Epoch 8/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0935 - acc: 0.9709 - val_
Epoch 9/20
60000/60000 [==============================] - 4s 75us/step - loss: 0.0895 - acc: 0.9722 - val_
Epoch 10/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.0828 - acc: 0.9738 - val_
Epoch 11/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0788 - acc: 0.9746 - val_
Epoch 12/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.0745 - acc: 0.9763 - val_
Epoch 13/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0704 - acc: 0.9782 - val_
```

```
Epoch 14/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0690 - acc: 0.9783 - val_
Epoch 15/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0669 - acc: 0.9793 - val_
Epoch 16/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0617 - acc: 0.9808 - val_
Epoch 17/20
60000/60000 [==============================] - 4s 75us/step - loss: 0.0621 - acc: 0.9807 - val_
Epoch 18/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0564 - acc: 0.9817 - val_
Epoch 19/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0575 - acc: 0.9817 - val_
Epoch 20/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0531 - acc: 0.9830 - val_
Test score: 0.054348885305505246
Test accuracy: 0.9829
```





3.2 Model 2 (3-layered MLP Architecture):

3.2.1 Plot Weights: Common Function

```
In [71]: def plotWeightM2(model):
             w_after = model.get_weights()

             h1_w = w_after[0].flatten().reshape(-1,1)
             h2_w = w_after[2].flatten().reshape(-1,1)
             h3_w = w_after[4].flatten().reshape(-1,1)
             out_w = w_after[6].flatten().reshape(-1,1)


             fig = plt.figure()
             plt.title("Weight matrices after model trained")
             plt.subplot(1, 4, 1)
             plt.title("Trained Wt")
             ax = sns.violinplot(y=h1_w,color='b')
             plt.xlabel('Hidden Layer 1')

             plt.subplot(1, 4, 2)
             plt.title("Trained Wt")
             ax = sns.violinplot(y=h2_w, color='r')
             plt.xlabel('Hidden Layer 2 ')
```

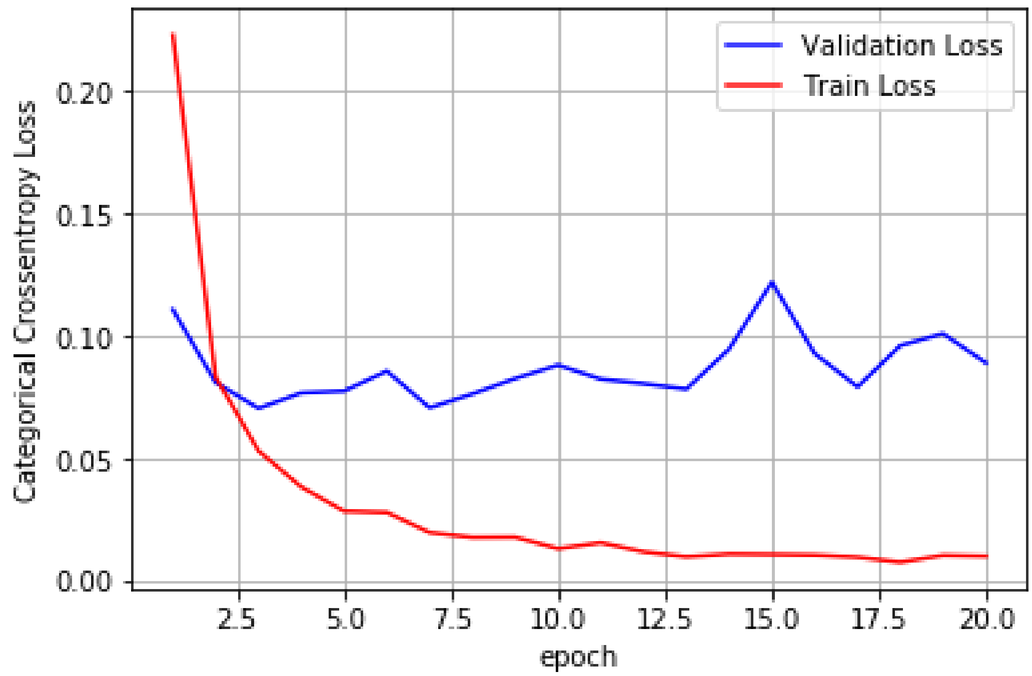
```
    plt.subplot(1, 4, 3)
    plt.title("Trained Wt")
    ax = sns.violinplot(y=h3_w, color='g')
    plt.xlabel('Hidden Layer 3 ')

    plt.subplot(1, 4, 4)
    plt.title("Trained Wt")
    ax = sns.violinplot(y=out_w,color='m')
    plt.xlabel('Output Layer ')
    plt.show()
```

3.2.2 Input(784)-ReLu(512)-ReLu(256)-ReLu(64)-Softmax(output)

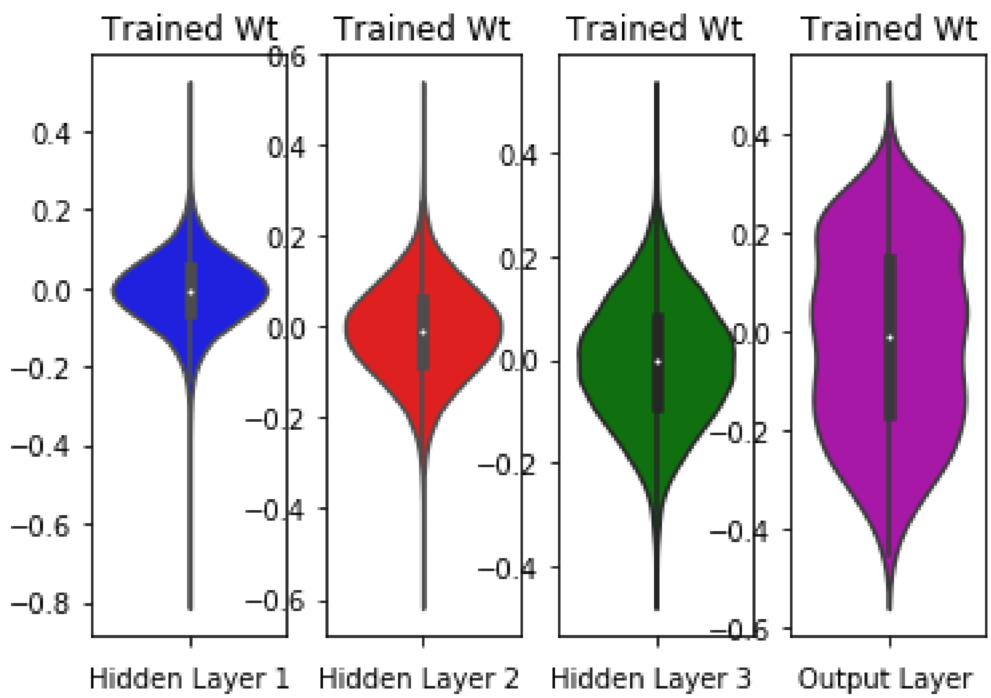
```
In [72]: model_relu = Sequential()
         model_relu.add(Dense(512, activation='relu',
                              input_shape=(input_dim,), kernel_initializer='he_normal'))
         model_relu.add(Dense(256, activation='relu', kernel_initializer='he_normal'))
         model_relu.add(Dense(64, activation='relu', kernel_initializer='he_normal'))
         model_relu.add(Dense(output_dim, activation='softmax'))

         print(model_relu.summary())

         history = trainModel(model=model_relu)
         plotGraph(model=model_relu, history=history)
         plotWeightM2(model=model_relu)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_88 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_89 (Dense)             (None, 256)               131328
_________________________________________________________________
dense_90 (Dense)             (None, 64)                16448
_________________________________________________________________
dense_91 (Dense)             (None, 10)                650
=================================================================
Total params: 550,346
Trainable params: 550,346
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 16s 269us/step - loss: 0.2229 - acc: 0.9336 - va
Epoch 2/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0834 - acc: 0.9749 - val_
Epoch 3/20
```

```
60000/60000 [==============================] - 4s 67us/step - loss: 0.0533 - acc: 0.9836 - val_
Epoch 4/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0386 - acc: 0.9874 - val_
Epoch 5/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0287 - acc: 0.9906 - val_
Epoch 6/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0282 - acc: 0.9912 - val_
Epoch 7/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0199 - acc: 0.9931 - val_
Epoch 8/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0180 - acc: 0.9942 - val_
Epoch 9/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0181 - acc: 0.9940 - val_
Epoch 10/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0134 - acc: 0.9957 - val_
Epoch 11/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0158 - acc: 0.9948 - val_
Epoch 12/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0121 - acc: 0.9961 - val_
Epoch 13/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0101 - acc: 0.9969 - val_
Epoch 14/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0112 - acc: 0.9963 - val_
Epoch 15/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0110 - acc: 0.9962 - val_
Epoch 16/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0108 - acc: 0.9966 - val_
Epoch 17/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0099 - acc: 0.9969 - val_
Epoch 18/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.0078 - acc: 0.9977 - val_
Epoch 19/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0106 - acc: 0.9967 - val_
Epoch 20/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0103 - acc: 0.9968 - val_
Test score: 0.08919506263012462
Test accuracy: 0.9807
```



Validation Loss
Train Loss
Categorical Crossentropy Loss
0.00
0.05
0.10
0.15
0.20
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
epoch



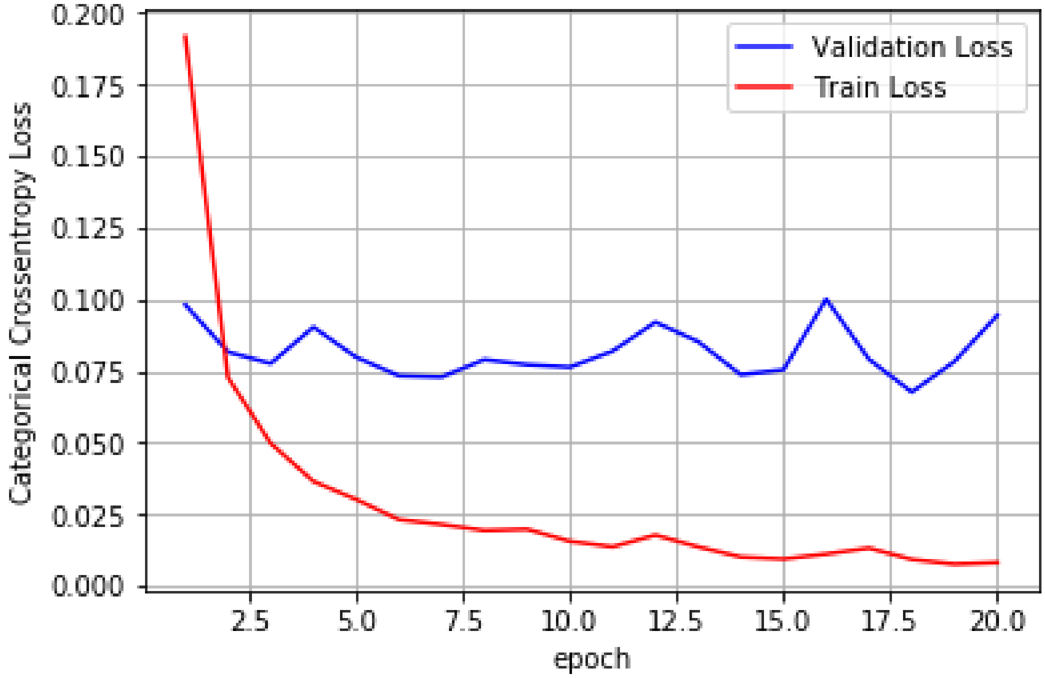
Trained Wt
Trained Wt
Trained Wt
Trained Wt
Hidden Layer 1
Hidden Layer 2
Hidden Layer 3
Output Layer

3.2.3 Model 2: M2 + Batch-Normalization on 3 hidden Layers

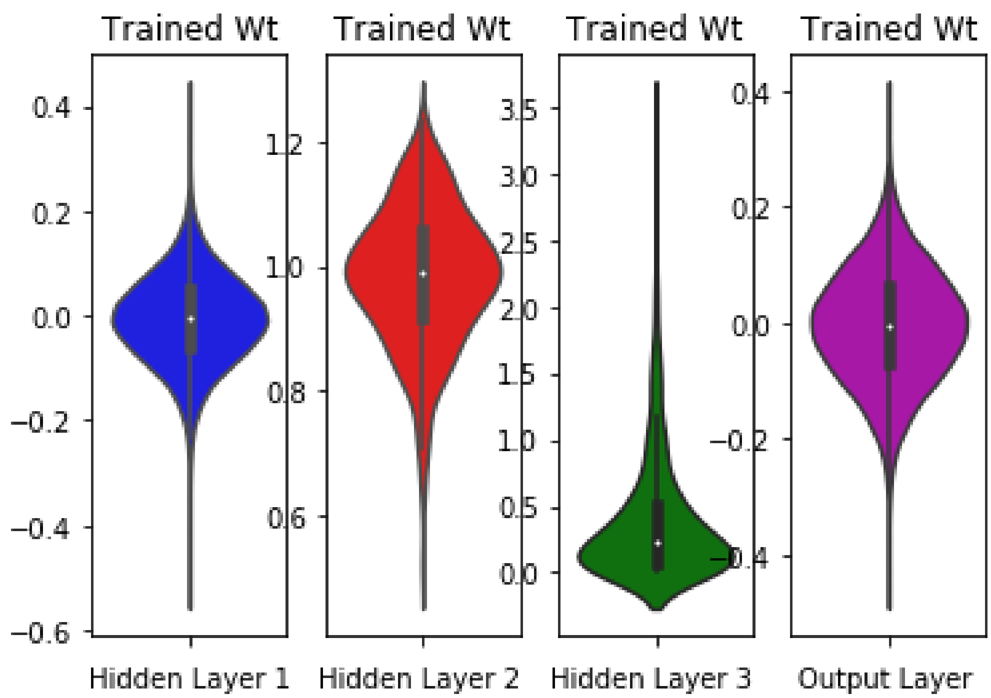
In [73]:
```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,) we satisfy this condition wit
# h1 =>  =(2/(ni+ni+1) = 0.039  => N(0,) = N(0,0.039)
# h2 =>  =(2/(ni+ni+1) = 0.055  => N(0,) = N(0,0.055)
# h1 =>  =(2/(ni+ni+1) = 0.120  => N(0,) = N(0,0.120)

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(512, activation='relu',
                      input_shape=(input_dim,), kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(256, activation='relu', kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(64, activation='relu', kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(output_dim, activation='softmax'))

model_batch.summary()

history = trainModel(model=model_batch)
plotGraph(model=model_batch, history=history)
plotWeightM2(model=model_batch)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_92 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_45 (Batc (None, 512)               2048
_________________________________________________________________
dense_93 (Dense)             (None, 256)               131328
_________________________________________________________________
batch_normalization_46 (Batc (None, 256)               1024
_________________________________________________________________
dense_94 (Dense)             (None, 64)                16448
_________________________________________________________________
batch_normalization_47 (Batc (None, 64)                256
_________________________________________________________________
dense_95 (Dense)             (None, 10)                650
=================================================================
```

```
Total params: 553,674
Trainable params: 552,010
Non-trainable params: 1,664
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 19s 313us/step - loss: 0.1918 - acc: 0.9438 - va
Epoch 2/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0727 - acc: 0.9782 - val_
Epoch 3/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0498 - acc: 0.9850 - val_
Epoch 4/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0365 - acc: 0.9885 - val_
Epoch 5/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0302 - acc: 0.9900 - val_
Epoch 6/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0232 - acc: 0.9928 - val_
Epoch 7/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0214 - acc: 0.9931 - val_
Epoch 8/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0194 - acc: 0.9936 - val_
Epoch 9/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.0198 - acc: 0.9936 - va
Epoch 10/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0156 - acc: 0.9952 - val_
Epoch 11/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0136 - acc: 0.9955 - val_
Epoch 12/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0177 - acc: 0.9940 - val_
Epoch 13/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0136 - acc: 0.9954 - val_
Epoch 14/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0101 - acc: 0.9969 - val_
Epoch 15/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0094 - acc: 0.9970 - val_
Epoch 16/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0111 - acc: 0.9964 - val_
Epoch 17/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0132 - acc: 0.9955 - val_
Epoch 18/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0092 - acc: 0.9969 - val_
Epoch 19/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0076 - acc: 0.9977 - val_
Epoch 20/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0081 - acc: 0.9973 - val_
Test score: 0.0944686686351095
Test accuracy: 0.9776
```



Validation Loss
Train Loss
Categorical Crossentropy Loss
epoch
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0



Trained Wt
Trained Wt
Trained Wt
Trained Wt
Hidden Layer 1
Hidden Layer 2
Hidden Layer 3
Output Layer

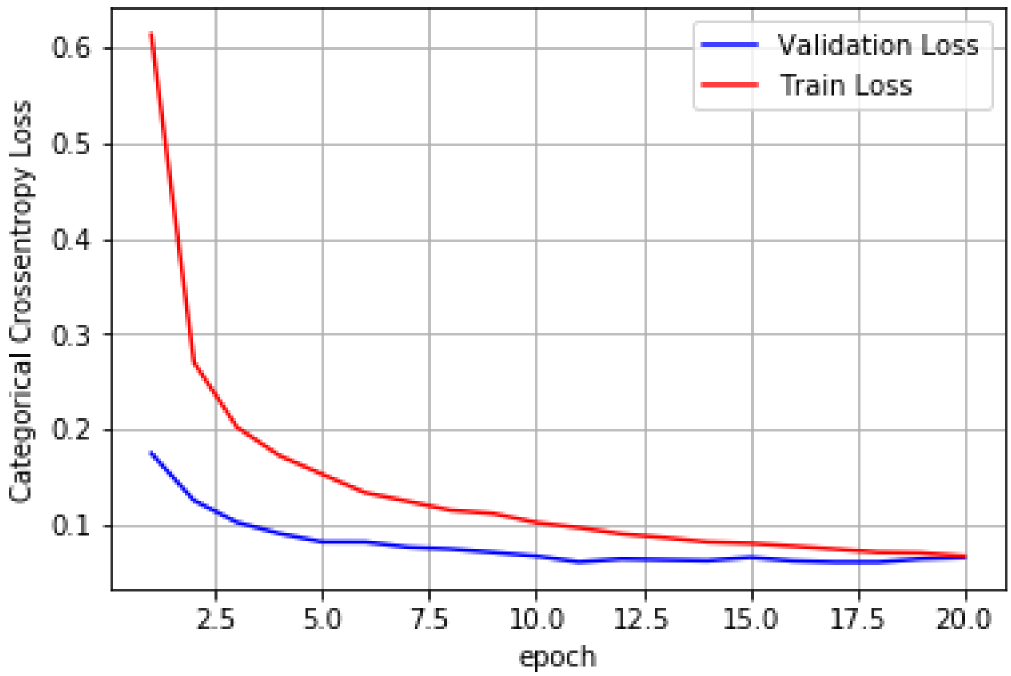
3.2.4 Model 2: M2 + Batch-Normalization + Dropout

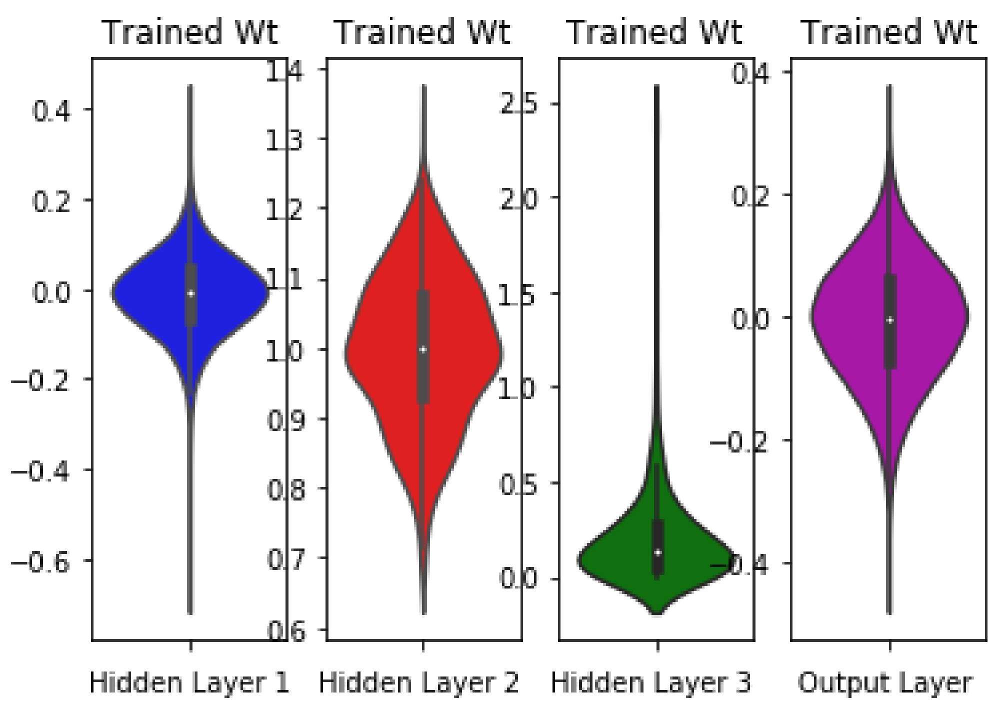
```
In [74]: # https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization

         from keras.layers import Dropout

         model_drop = Sequential()

         model_drop.add(Dense(512, activation='relu',
                         input_shape=(input_dim,), kernel_initializer='he_normal'))
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))

         model_drop.add(Dense(256, activation='relu', kernel_initializer='he_normal'))
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))

         model_drop.add(Dense(64, activation='relu', kernel_initializer='he_normal'))
         model_drop.add(BatchNormalization())
         model_drop.add(Dropout(0.5))

         model_drop.add(Dense(output_dim, activation='softmax'))

         model_drop.summary()

         history = trainModel(model=model_drop)
         plotGraph(model=model_drop, history=history)
         plotWeightM2(model=model_drop)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_96 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_48 (Batc (None, 512)               2048
_________________________________________________________________
dropout_23 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_97 (Dense)             (None, 256)               131328
_________________________________________________________________
batch_normalization_49 (Batc (None, 256)               1024
_________________________________________________________________
dropout_24 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_98 (Dense)             (None, 64)                16448
_________________________________________________________________
batch_normalization_50 (Batc (None, 64)                256
_________________________________________________________________
dropout_25 (Dropout)         (None, 64)                0
```

```
_________________________________________________________________
dense_99 (Dense)             (None, 10)                650
=================================================================
Total params: 553,674
Trainable params: 552,010
Non-trainable params: 1,664
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 19s 310us/step - loss: 0.6132 - acc: 0.8129 - va
Epoch 2/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.2699 - acc: 0.9229 - val_
Epoch 3/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.2023 - acc: 0.9429 - val_
Epoch 4/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.1724 - acc: 0.9514 - val_
Epoch 5/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1530 - acc: 0.9574 - val_
Epoch 6/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1337 - acc: 0.9614 - val_
Epoch 7/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1246 - acc: 0.9656 - val_
Epoch 8/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1153 - acc: 0.9673 - val_
Epoch 9/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1118 - acc: 0.9674 - val_
Epoch 10/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.1024 - acc: 0.9708 - val_
Epoch 11/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0969 - acc: 0.9715 - val_
Epoch 12/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0904 - acc: 0.9744 - val_
Epoch 13/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0866 - acc: 0.9748 - val_
Epoch 14/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0820 - acc: 0.9759 - val_
Epoch 15/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0806 - acc: 0.9767 - val_
Epoch 16/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0777 - acc: 0.9769 - val_
Epoch 17/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0745 - acc: 0.9775 - val_
Epoch 18/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0712 - acc: 0.9787 - val_
Epoch 19/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0706 - acc: 0.9792 - val_
Epoch 20/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0672 - acc: 0.9801 - val_
```

```
Test score: 0.06557027021200047
Test accuracy: 0.9828
```





3.3 Model 3 (5-layered MLP Architecture):

3.3.1 Plot Weights: Common Function

```
In [75]: def plotWeightM3(model):
             w_after = model.get_weights()

             h1_w = w_after[0].flatten().reshape(-1,1)
             h2_w = w_after[2].flatten().reshape(-1,1)
             h3_w = w_after[4].flatten().reshape(-1,1)
             h4_w = w_after[6].flatten().reshape(-1,1)
             h5_w = w_after[8].flatten().reshape(-1,1)
             out_w = w_after[10].flatten().reshape(-1,1)


             fig = plt.figure()
             plt.title("Weight matrices after model trained")
             plt.subplot(1, 6, 1)
             plt.title("Weights")
             ax = sns.violinplot(y=h1_w,color='b')
             plt.xlabel('Hidden 1')

             plt.subplot(1, 6, 2)
             plt.title("Weights")
             ax = sns.violinplot(y=h2_w, color='r')
             plt.xlabel('Hidden 2 ')

             plt.subplot(1, 6, 3)
             plt.title("Weights")
             ax = sns.violinplot(y=h3_w, color='g')
             plt.xlabel('Hidden 3 ')

             plt.subplot(1, 6, 4)
             plt.title("Weights")
             ax = sns.violinplot(y=h4_w, color='c')
             plt.xlabel('Hidden 4 ')

             plt.subplot(1, 6, 5)
             plt.title("Weights")
             ax = sns.violinplot(y=h5_w, color='m')
             plt.xlabel('Hidden 5 ')

             plt.subplot(1, 6, 6)
             plt.title("Weights")
             ax = sns.violinplot(y=out_w,color='y')
```

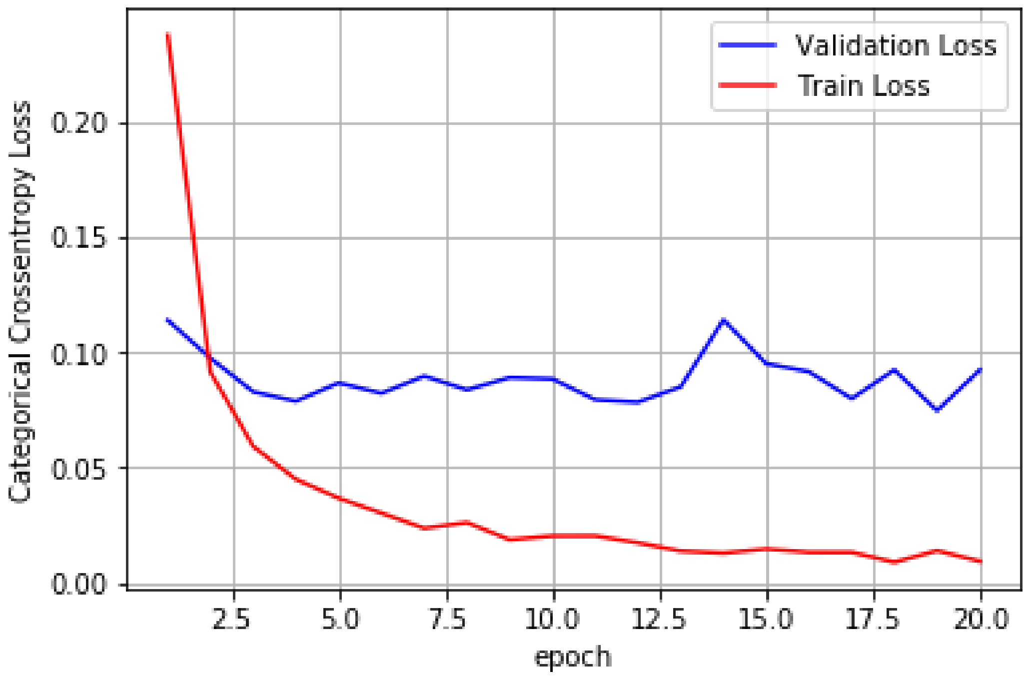
```
    plt.xlabel('Out Layer')
    plt.show()
```

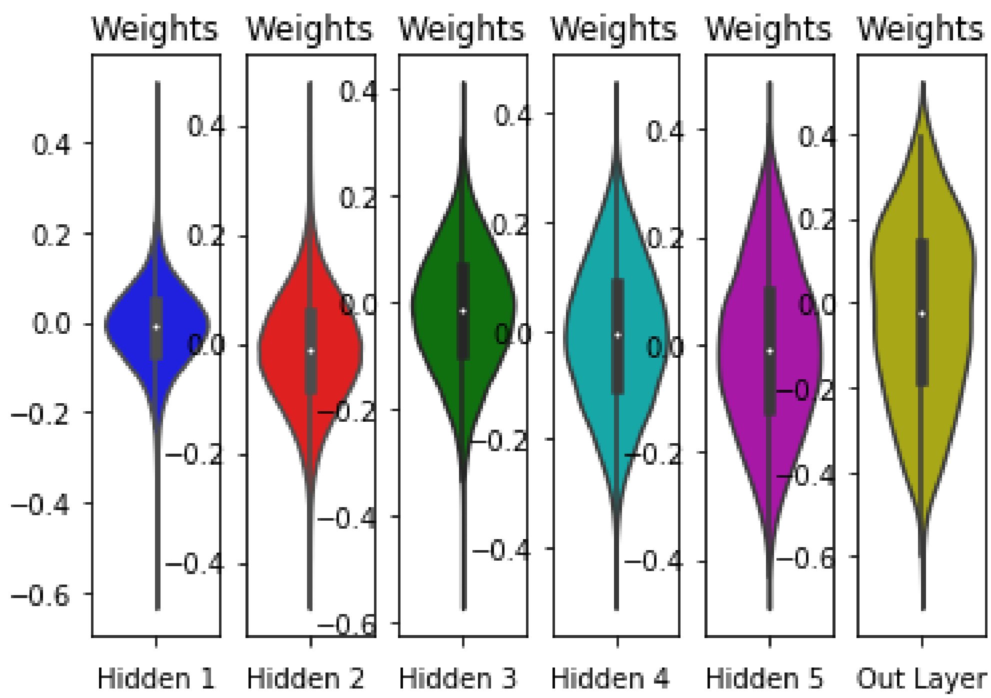
3.3.2 Input(784)-ReLu(512)-ReLu(256)-ReLu(144)-ReLu(96)-ReLu(36)-Softmax(output)

```
In [76]: model_relu = Sequential()
         model_relu.add(Dense(512, activation='relu',
                        input_shape=(input_dim,), kernel_initializer='he_normal'))
         model_relu.add(Dense(256, activation='relu', kernel_initializer='he_normal'))
         model_relu.add(Dense(144, activation='relu', kernel_initializer='he_normal'))
         model_relu.add(Dense(96, activation='relu', kernel_initializer='he_normal'))
         model_relu.add(Dense(36, activation='relu', kernel_initializer='he_normal'))
         model_relu.add(Dense(output_dim, activation='softmax'))

         print(model_relu.summary())

         history = trainModel(model=model_relu)
         plotGraph(model=model_relu, history=history)
         plotWeightM3(model=model_relu)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_100 (Dense)            (None, 512)               401920
_________________________________________________________________
dense_101 (Dense)            (None, 256)               131328
_________________________________________________________________
dense_102 (Dense)            (None, 144)               37008
_________________________________________________________________
dense_103 (Dense)            (None, 96)                13920
_________________________________________________________________
dense_104 (Dense)            (None, 36)                3492
_________________________________________________________________
dense_105 (Dense)            (None, 10)                370
=================================================================
Total params: 588,038
Trainable params: 588,038
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 17s 286us/step - loss: 0.2376 - acc: 0.9294 - va
Epoch 2/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0914 - acc: 0.9722 - val_
Epoch 3/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0592 - acc: 0.9814 - val_
Epoch 4/20
```

```
60000/60000 [==============================] - 5s 77us/step - loss: 0.0449 - acc: 0.9856 - val_
Epoch 5/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0368 - acc: 0.9880 - val_
Epoch 6/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.0303 - acc: 0.9900 - val_
Epoch 7/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0237 - acc: 0.9921 - val_
Epoch 8/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0262 - acc: 0.9919 - val_
Epoch 9/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0187 - acc: 0.9940 - val_
Epoch 10/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0204 - acc: 0.9938 - val_
Epoch 11/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0204 - acc: 0.9934 - val_
Epoch 12/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0174 - acc: 0.9944 - val_
Epoch 13/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0138 - acc: 0.9955 - val_
Epoch 14/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0129 - acc: 0.9955 - val_
Epoch 15/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0147 - acc: 0.9954 - val_
Epoch 16/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0133 - acc: 0.9960 - val_
Epoch 17/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0133 - acc: 0.9960 - val_
Epoch 18/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0090 - acc: 0.9971 - val_
Epoch 19/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0140 - acc: 0.9960 - val_
Epoch 20/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0094 - acc: 0.9972 - val_
Test score: 0.09236639852523267
Test accuracy: 0.9818
```


Validation Loss
Train Loss
Categorical Crossentropy Loss
0.20
0.15
0.10
0.05
0.00
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
epoch


Weights
Weights
Weights
Weights
Weights
Weights
Hidden 1
Hidden 2
Hidden 3
Hidden 4
Hidden 5
Out Layer

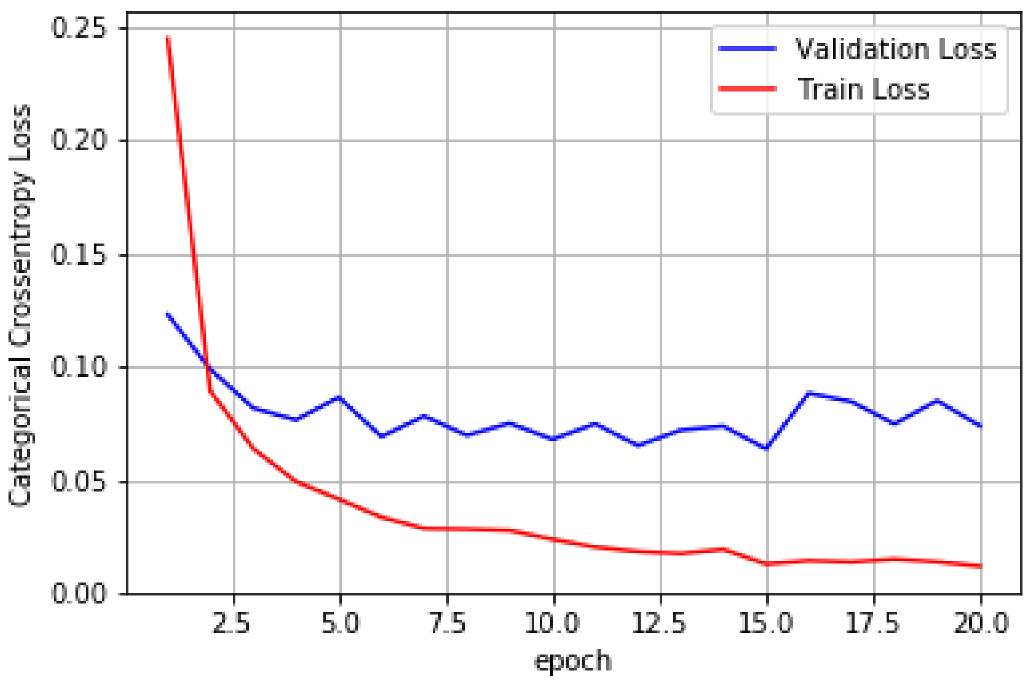
3.3.3 Model 3: M3 + Batch-Normalization on 5 hidden Layers

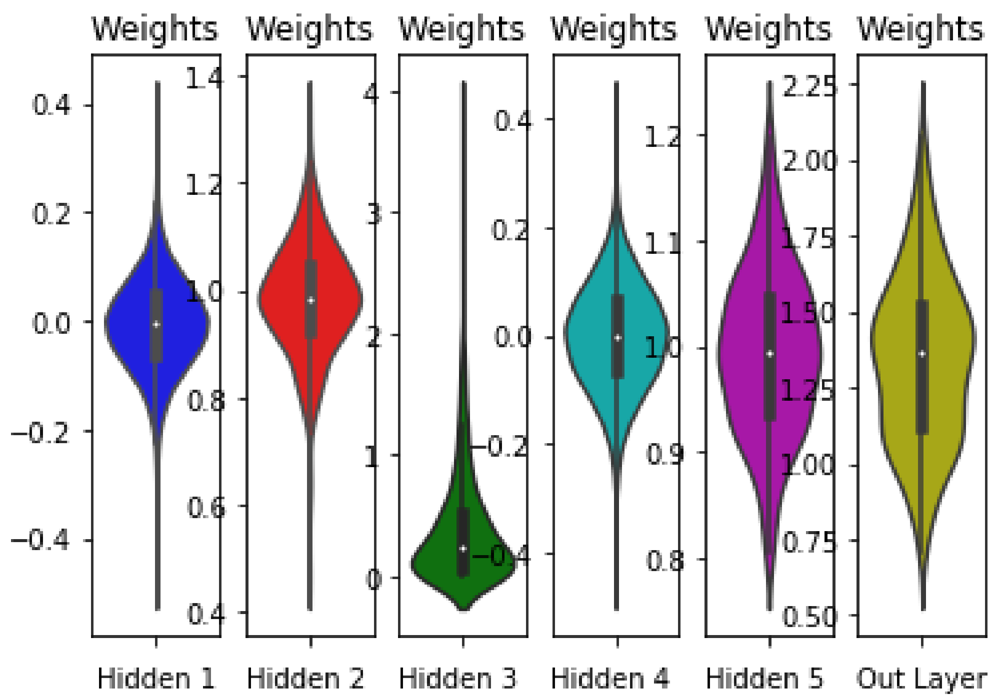
In [77]:
```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,) we satisfy this condition wit
# h1 =>  =(2/(ni+ni+1) = 0.039  => N(0,) = N(0,0.039)
# h2 =>  =(2/(ni+ni+1) = 0.055  => N(0,) = N(0,0.055)
# h1 =>  =(2/(ni+ni+1) = 0.120  => N(0,) = N(0,0.120)

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(512, activation='relu',
                    input_shape=(input_dim,), kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(256, activation='relu', kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(144, activation='relu', kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(96, activation='relu', kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(36, activation='relu', kernel_initializer='he_normal'))
model_batch.add(BatchNormalization())

model_batch.add(Dense(output_dim, activation='softmax'))

model_batch.summary()


history = trainModel(model=model_batch)
plotGraph(model=model_batch, history=history)
plotWeightM3(model=model_batch)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_106 (Dense)            (None, 512)               401920
_________________________________________________________________
batch_normalization_51 (Batc (None, 512)               2048
_________________________________________________________________
dense_107 (Dense)            (None, 256)               131328
_________________________________________________________________
batch_normalization_52 (Batc (None, 256)               1024
```

```
_________________________________________________________________
dense_108 (Dense)            (None, 144)              37008
_________________________________________________________________
batch_normalization_53 (Batc (None, 144)              576
_________________________________________________________________
dense_109 (Dense)            (None, 96)               13920
_________________________________________________________________
batch_normalization_54 (Batc (None, 96)               384
_________________________________________________________________
dense_110 (Dense)            (None, 36)               3492
_________________________________________________________________
batch_normalization_55 (Batc (None, 36)               144
_________________________________________________________________
dense_111 (Dense)            (None, 10)               370
=================================================================
Total params: 592,214
Trainable params: 590,126
Non-trainable params: 2,088
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 19s 325us/step - loss: 0.2446 - acc: 0.9314 - va
Epoch 2/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0890 - acc: 0.9734 - val_
Epoch 3/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0637 - acc: 0.9801 - val_
Epoch 4/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.0492 - acc: 0.9849 - val_
Epoch 5/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.0414 - acc: 0.9865 - val_
Epoch 6/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0334 - acc: 0.9895 - val_
Epoch 7/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0285 - acc: 0.9904 - val_
Epoch 8/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0282 - acc: 0.9903 - val
Epoch 9/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0276 - acc: 0.9908 - val
Epoch 10/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0237 - acc: 0.9923 - val_
Epoch 11/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0202 - acc: 0.9936 - val_
Epoch 12/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0183 - acc: 0.9938 - val_
Epoch 13/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0174 - acc: 0.9943 - val
Epoch 14/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.0192 - acc: 0.9937 - val
```

```
Epoch 15/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0128 - acc: 0.9959 - val
Epoch 16/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0142 - acc: 0.9956 - val
Epoch 17/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.0137 - acc: 0.9957 - val
Epoch 18/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.0149 - acc: 0.9951 - val
Epoch 19/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.0137 - acc: 0.9956 - val_
Epoch 20/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0118 - acc: 0.9961 - val_
Test score: 0.07374950621149037
Test accuracy: 0.9813
```





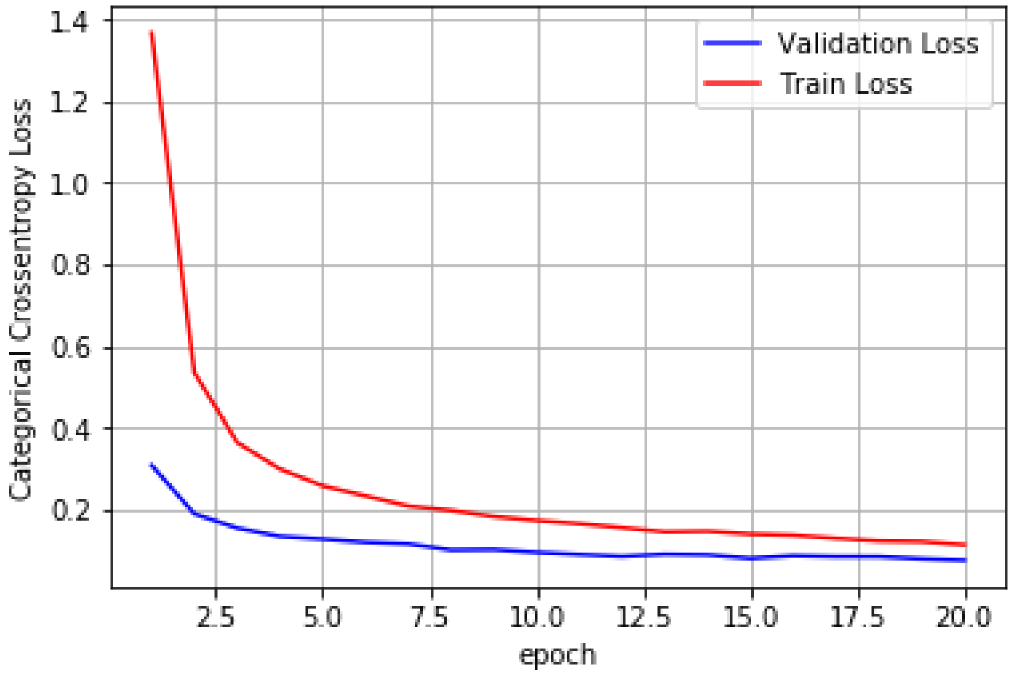
3.3.4 Model 3: M3 + Batch-Normalization + Dropout

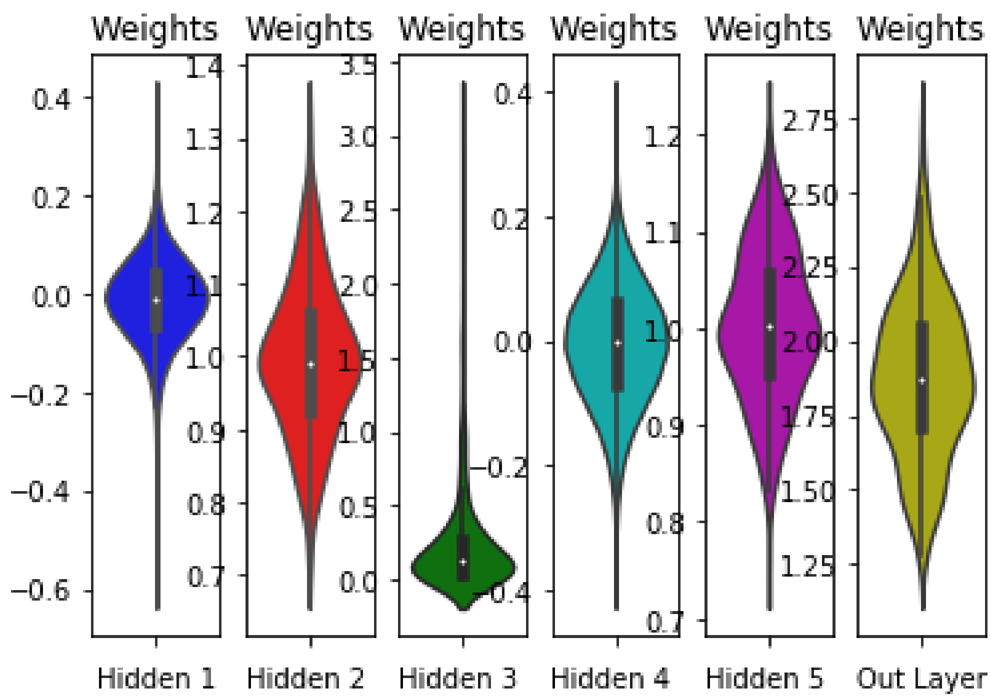
In [78]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='relu',
                input_shape=(input_dim,), kernel_initializer='he_normal'))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(256, activation='relu', kernel_initializer='he_normal'))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(144, activation='relu', kernel_initializer='he_normal'))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(96, activation='relu', kernel_initializer='he_normal'))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))
```

```python
model_drop.add(Dense(36, activation='relu', kernel_initializer='he_normal'))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(output_dim, activation='softmax'))

model_drop.summary()

history = trainModel(model=model_drop)
plotGraph(model=model_drop, history=history)
plotWeightM3(model=model_drop)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_112 (Dense)            (None, 512)               401920
_________________________________________________________________
batch_normalization_56 (Batc (None, 512)               2048
_________________________________________________________________
dropout_26 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_113 (Dense)            (None, 256)               131328
_________________________________________________________________
batch_normalization_57 (Batc (None, 256)               1024
_________________________________________________________________
dropout_27 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_114 (Dense)            (None, 144)               37008
_________________________________________________________________
batch_normalization_58 (Batc (None, 144)               576
_________________________________________________________________
dropout_28 (Dropout)         (None, 144)               0
_________________________________________________________________
dense_115 (Dense)            (None, 96)                13920
_________________________________________________________________
batch_normalization_59 (Batc (None, 96)                384
_________________________________________________________________
dropout_29 (Dropout)         (None, 96)                0
_________________________________________________________________
dense_116 (Dense)            (None, 36)                3492
_________________________________________________________________
batch_normalization_60 (Batc (None, 36)                144
_________________________________________________________________
dropout_30 (Dropout)         (None, 36)                0
_________________________________________________________________
dense_117 (Dense)            (None, 10)                370
=================================================================
```

```
Total params: 592,214
Trainable params: 590,126
Non-trainable params: 2,088
-----------------------------------------------------------------
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 20s 341us/step - loss: 1.3673 - acc: 0.5682 - va
Epoch 2/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.5358 - acc: 0.8528 - val
Epoch 3/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.3650 - acc: 0.9085 - val
Epoch 4/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.3002 - acc: 0.9260 - val
Epoch 5/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.2589 - acc: 0.9363 - val
Epoch 6/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.2342 - acc: 0.9440 - val
Epoch 7/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.2094 - acc: 0.9494 - val
Epoch 8/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.1986 - acc: 0.9531 - val
Epoch 9/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.1835 - acc: 0.9567 - val
Epoch 10/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.1738 - acc: 0.9589 - val
Epoch 11/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.1660 - acc: 0.9616 - val
Epoch 12/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.1564 - acc: 0.9627 - val
Epoch 13/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.1461 - acc: 0.9659 - val
Epoch 14/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.1475 - acc: 0.9655 - val
Epoch 15/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.1407 - acc: 0.9672 - val
Epoch 16/20
60000/60000 [==============================] - 7s 108us/step - loss: 0.1378 - acc: 0.9675 - val
Epoch 17/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.1303 - acc: 0.9687 - val
Epoch 18/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.1238 - acc: 0.9712 - val
Epoch 19/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.1223 - acc: 0.9719 - val
Epoch 20/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.1145 - acc: 0.9733 - val
Test score: 0.0764596716644708
Test accuracy: 0.9814
```


Validation Loss
Train Loss
Categorical Crossentropy Loss
epoch
1.4
1.2
1.0
0.8
0.6
0.4
0.2
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Weights
Weights
Weights
Weights
Weights
Weights
Hidden 1
Hidden 2
Hidden 3
Hidden 4
Hidden 5
Out Layer

4 Summary Statistics

Model	Total Parameters	Accuracy	Loss vs Epoch Plot
Model 1 (784x512x128x10)	468,874	**98.18**	Diverging
M1 + Batch-Normalization	471,434	**98.06**	Diverging
M1 + Batch-Normalization+Dropout	471,434	**98.32**	**Converging**
Model 2 (784x512x256x64x10)	550,346	**97.85**	Diverging
M2 + Batch-Normalization	553,674	**98.06**	Diverging
M2 + Batch-Normalization+Dropout	553,674	**98.24**	**Converging**
Model 3 (784x512x256x144x96x10)	588,038	**97.96**	Diverging
M3 + Batch-Normalization	592,214	**97.88**	Diverging
M3 + Batch-Normalization+Dropout	592,214	**98.26**	**Converging**

5 Conclusions

1. The **difference in accuracy between 2, 3 & 5 layered networks is very small**. This could be due to the simplicity and small size of input data.
2. The **'cross entropy loss vs epoch' plot** for train and test data is found **diverging, when the dropout layer is not added**. This means reduction in training loss but increase in test loss at the same time, **indicative of overfitting.**
3. Thus, **addition of dropout layer is found as a good regularization** in practice.
4. The accuracy is also more when Batch Normalization and dropout layers are added.
5. All distributions of trained weights along all the layers are expectedly found as Gaussian curves.
6. For MNIST problem, **model M1 coupled with Batch Normalizaton and Dropout seems to be the best bet.**