Keras -- MLPs on MNIST

```
# 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
```

```
%matplotlib inline
%matplotlib notebook
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(x, vy, ty, fig, 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()
```

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

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

```
# if you observe the input shape its 2 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])
```

```
# 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 training 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 training examples : 10000 and each image is of shape (784)
```

```
# 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  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   0
  82  56  39   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   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]
```

```
# 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
```

```
# 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.
```

```
0.99215686 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.        ]
```

```python
# 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.]
```

Softmax classifier

```python
# 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'),
# ])

# 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 ar available ex: tanh, relu, softmax


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

```

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

```
%matplotlib inline
def loss_plot(model_name):
    score = model_name.evaluate(X_test, Y_test, verbose=0)
    print('Test score:', score[0])
    print('Test accuracy:', score[1])

    fig, ax = plt.subplots(1,1, figsize=(10,6))
    ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentropy Loss')
    ax.set_title('Variation of Loss with epochs')

    # 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(x, vy, ty, fig, ax)
```

```
def weight_plot(model_name):
    w_after = model_name.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(figsize=(15,7))
    fig.suptitle("Weight matrices after model trained")
    plt.subplot(1, 3, 1)
    ax = sns.violinplot(y=h1_w,color='b')
    plt.xlabel('Hidden Layer 1')

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

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

Architecture 1 : Layer1(600), Layer2(100)

i) MLP + ReLU + ADAM

```
import warnings
warnings.filterwarnings("ignore")
```

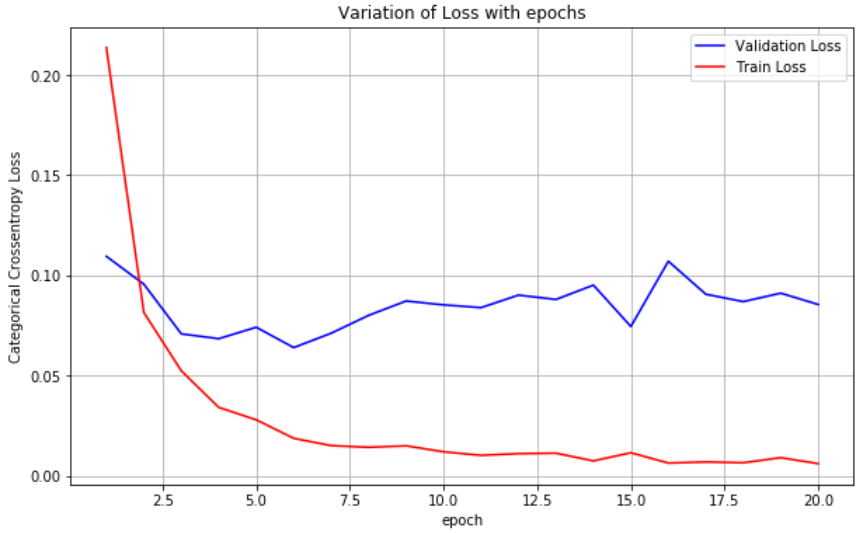
```
model_relu1 = Sequential()
```

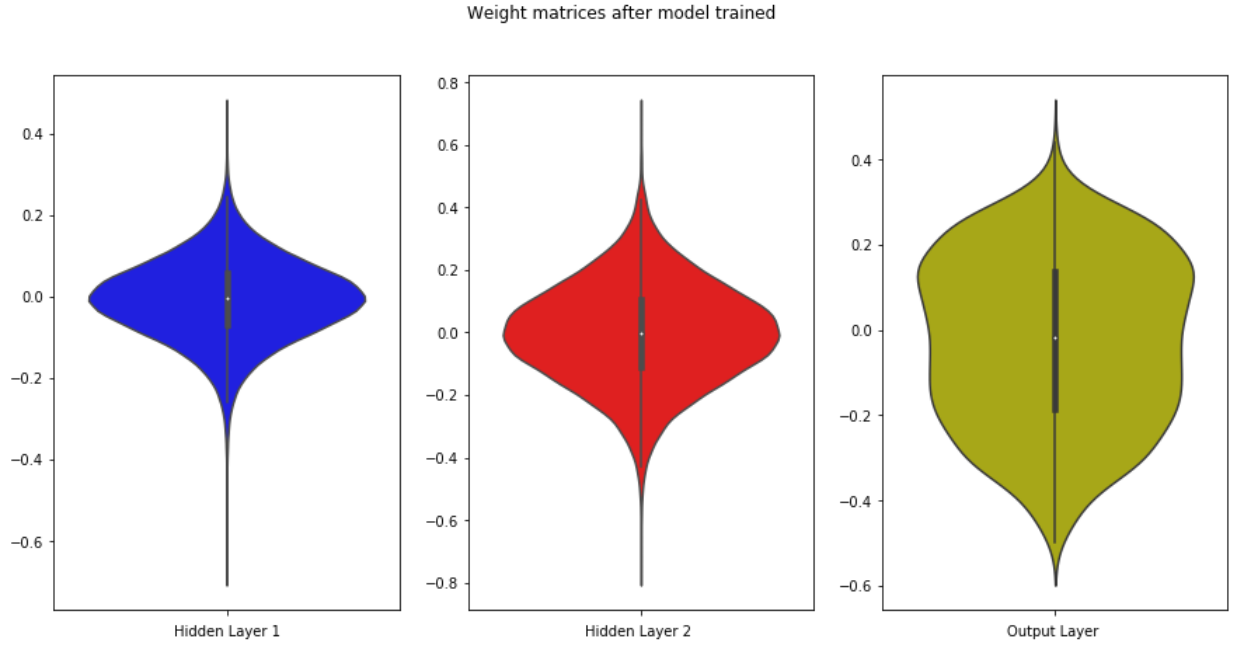
```
model_relu1.add(Dense(600, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.062, seed=Nor
model_relu1.add(Dense(100, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)) )
model_relu1.add(Dense(output_dim, activation='softmax'))

print(model_relu1.summary())

model_relu1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu1.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_42 (Dense)             (None, 600)               471000
_________________________________________________________________
dense_43 (Dense)             (None, 100)               60100
_________________________________________________________________
dense_44 (Dense)             (None, 10)                1010
=================================================================
Total params: 532,110
Trainable params: 532,110
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.2137 - acc: 0.9351 - val_loss: 0.1095 - val_acc: 0.9665
Epoch 2/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0815 - acc: 0.9755 - val_loss: 0.0956 - val_acc: 0.9713
Epoch 3/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0523 - acc: 0.9841 - val_loss: 0.0708 - val_acc: 0.9760
Epoch 4/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0341 - acc: 0.9890 - val_loss: 0.0684 - val_acc: 0.9798
Epoch 5/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0279 - acc: 0.9913 - val_loss: 0.0741 - val_acc: 0.9779
Epoch 6/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0187 - acc: 0.9941 - val_loss: 0.0639 - val_acc: 0.9809
Epoch 7/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0150 - acc: 0.9954 - val_loss: 0.0711 - val_acc: 0.9788
Epoch 8/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0142 - acc: 0.9957 - val_loss: 0.0800 - val_acc: 0.9794
Epoch 9/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0149 - acc: 0.9948 - val_loss: 0.0872 - val_acc: 0.9779
Epoch 10/20
60000/60000 [==============================] - 4s 58us/step - loss: 0.0119 - acc: 0.9960 - val_loss: 0.0853 - val_acc: 0.9800
Epoch 11/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0102 - acc: 0.9964 - val_loss: 0.0839 - val_acc: 0.9796
Epoch 12/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0110 - acc: 0.9962 - val_loss: 0.0901 - val_acc: 0.9780
Epoch 13/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0113 - acc: 0.9962 - val_loss: 0.0879 - val_acc: 0.9795
Epoch 14/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0074 - acc: 0.9975 - val_loss: 0.0951 - val_acc: 0.9791
Epoch 15/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0115 - acc: 0.9965 - val_loss: 0.0745 - val_acc: 0.9822
Epoch 16/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0063 - acc: 0.9979 - val_loss: 0.1070 - val_acc: 0.9766
Epoch 17/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0069 - acc: 0.9978 - val_loss: 0.0906 - val_acc: 0.9803
Epoch 18/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0065 - acc: 0.9979 - val_loss: 0.0869 - val_acc: 0.9806
Epoch 19/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0090 - acc: 0.9972 - val_loss: 0.0911 - val_acc: 0.9809
Epoch 20/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0060 - acc: 0.9980 - val_loss: 0.0855 - val_acc: 0.9838
```

```
loss_plot(model_relu1)
weight_plot(model_relu1)
```

```
Test score: 0.08545361435633295
Test accuracy: 0.9838
```





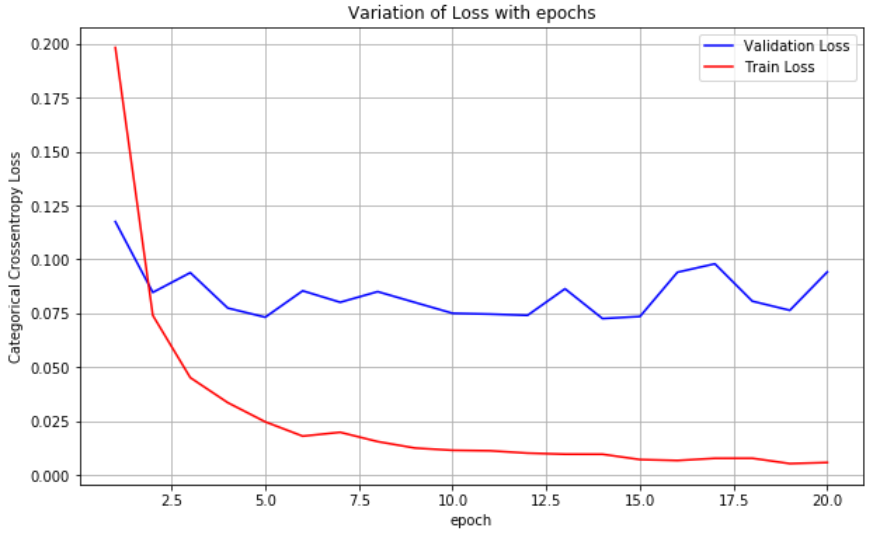
ii) MLP + Batch-Norm on hidden Layers + AdamOptimizer

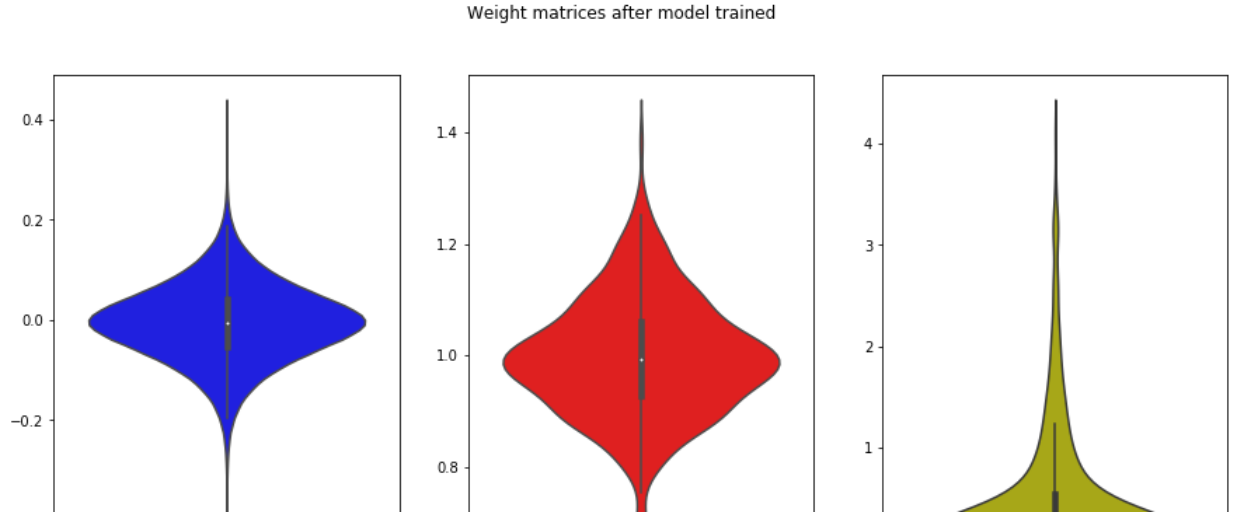
```python
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch1 = Sequential()

model_batch1.add(Dense(600, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=No
model_batch1.add(BatchNormalization())

model_batch1.add(Dense(100, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_batch1.add(BatchNormalization())

model_batch1.add(Dense(output_dim, activation='softmax'))


model_batch1.summary()
```

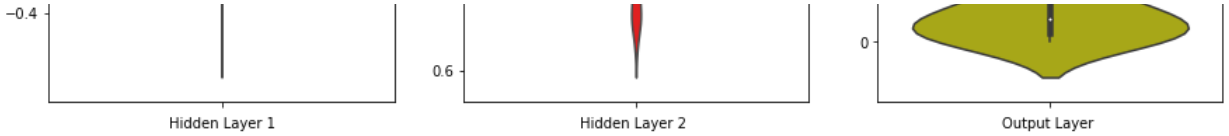
```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_45 (Dense)             (None, 600)               471000
_________________________________________________________________
batch_normalization_17 (Batc (None, 600)               2400
_________________________________________________________________
dense_46 (Dense)             (None, 100)               60100
_________________________________________________________________
batch_normalization_18 (Batc (None, 100)               400
_________________________________________________________________
dense_47 (Dense)             (None, 10)                1010
=================================================================
Total params: 534,910
Trainable params: 533,510
Non-trainable params: 1,400
_________________________________________________________________
```

```
model_batch1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_batch1.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
loss_plot(model_batch1)
weight_plot(model_batch1)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.1981 - acc: 0.9432 - val_loss: 0.1175 - val_acc: 0.961
Epoch 2/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0740 - acc: 0.9783 - val_loss: 0.0847 - val_acc: 0.9719
Epoch 3/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.0452 - acc: 0.9870 - val_loss: 0.0938 - val_acc: 0.969
Epoch 4/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0336 - acc: 0.9897 - val_loss: 0.0774 - val_acc: 0.975
Epoch 5/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0247 - acc: 0.9926 - val_loss: 0.0732 - val_acc: 0.9788
Epoch 6/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.0181 - acc: 0.9946 - val_loss: 0.0854 - val_acc: 0.975
Epoch 7/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0199 - acc: 0.9939 - val_loss: 0.0801 - val_acc: 0.975
Epoch 8/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.0156 - acc: 0.9955 - val_loss: 0.0850 - val_acc: 0.977
Epoch 9/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.0126 - acc: 0.9962 - val_loss: 0.0800 - val_acc: 0.975
Epoch 10/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0115 - acc: 0.9965 - val_loss: 0.0750 - val_acc: 0.978
Epoch 11/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.0113 - acc: 0.9964 - val_loss: 0.0746 - val_acc: 0.980
Epoch 12/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.0102 - acc: 0.9969 - val_loss: 0.0741 - val_acc: 0.980
Epoch 13/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.0097 - acc: 0.9970 - val_loss: 0.0864 - val_acc: 0.977
Epoch 14/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0097 - acc: 0.9969 - val_loss: 0.0726 - val_acc: 0.981
Epoch 15/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0072 - acc: 0.9975 - val_loss: 0.0735 - val_acc: 0.979
Epoch 16/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.0068 - acc: 0.9979 - val_loss: 0.0940 - val_acc: 0.975
Epoch 17/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.0078 - acc: 0.9975 - val_loss: 0.0979 - val_acc: 0.976
Epoch 18/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0078 - acc: 0.9975 - val_loss: 0.0806 - val_acc: 0.979
Epoch 19/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.0054 - acc: 0.9984 - val_loss: 0.0764 - val_acc: 0.982
Epoch 20/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.0059 - acc: 0.9981 - val_loss: 0.0942 - val_acc: 0.979
Test score: 0.09417202328130224
Test accuracy: 0.9796
```







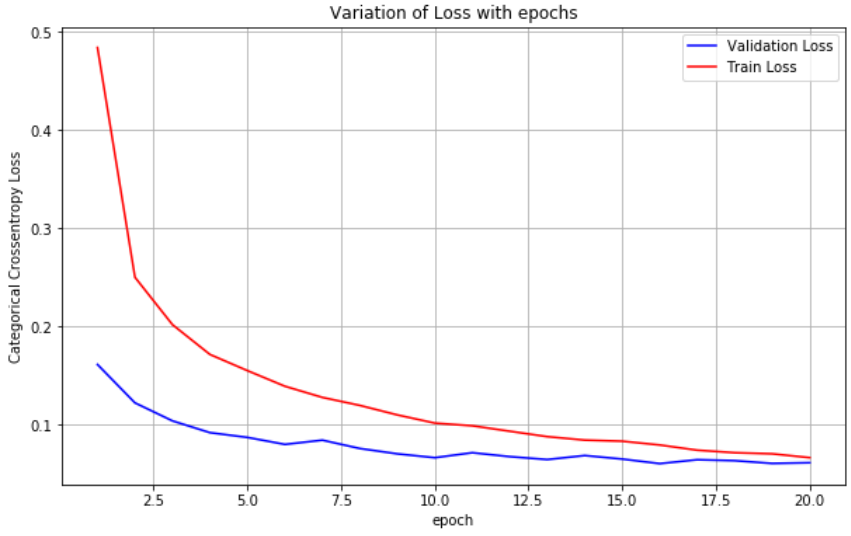
iii) MLP + Dropout + AdamOptimizer

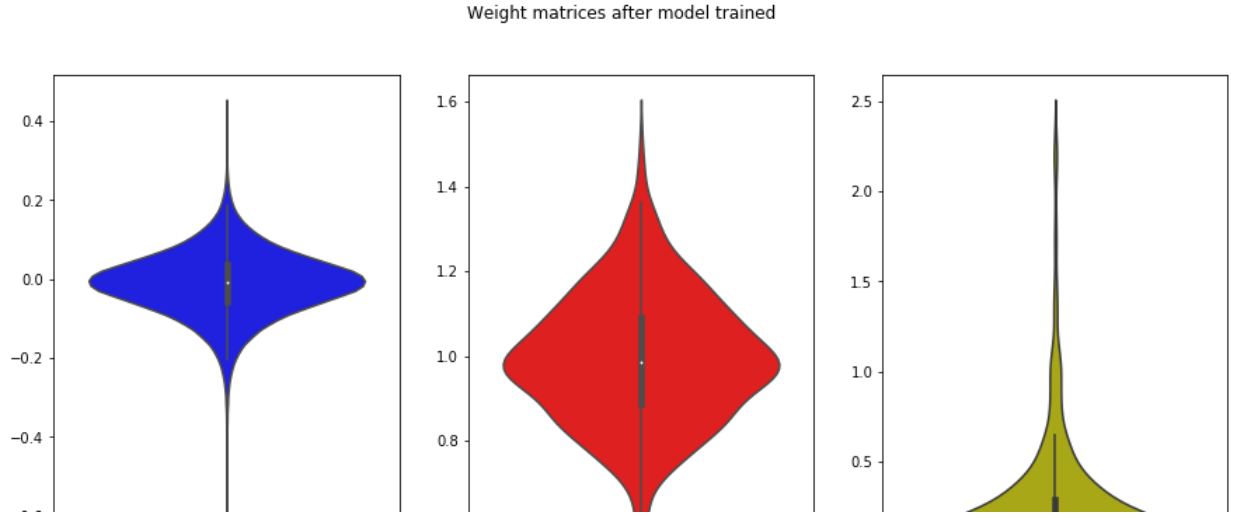
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization-function-in-keras

from keras.layers import Dropout

model_drop1 = Sequential()

model_drop1.add(Dense(600, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=Nor
model_drop1.add(BatchNormalization())
model_drop1.add(Dropout(0.5))

model_drop1.add(Dense(100, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop1.add(BatchNormalization())
model_drop1.add(Dropout(0.5))

model_drop1.add(Dense(output_dim, activation='softmax'))


model_drop1.summary()
```

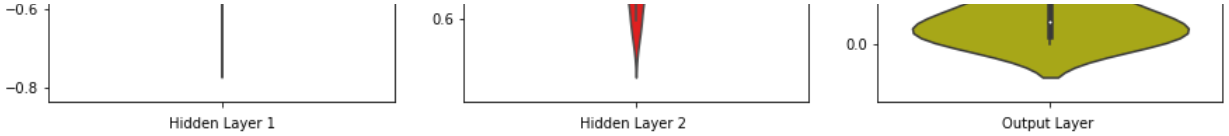
```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_48 (Dense)             (None, 600)               471000
_________________________________________________________________
batch_normalization_19 (Batc (None, 600)               2400
_________________________________________________________________
dropout_15 (Dropout)         (None, 600)               0
_________________________________________________________________
dense_49 (Dense)             (None, 100)               60100
_________________________________________________________________
batch_normalization_20 (Batc (None, 100)               400
_________________________________________________________________
dropout_16 (Dropout)         (None, 100)               0
_________________________________________________________________
dense_50 (Dense)             (None, 10)                1010
=================================================================
Total params: 534,910
Trainable params: 533,510
Non-trainable params: 1,400
_________________________________________________________________
```

```
model_drop1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_drop1.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
loss_plot(model_drop1)
weight_plot(model_drop1)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 10s 164us/step - loss: 0.4840 - acc: 0.8543 - val_loss: 0.1615 - val_acc: 0.95
Epoch 2/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.2502 - acc: 0.9259 - val_loss: 0.1224 - val_acc: 0.9597
Epoch 3/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.2020 - acc: 0.9401 - val_loss: 0.1041 - val_acc: 0.9685
Epoch 4/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.1717 - acc: 0.9490 - val_loss: 0.0921 - val_acc: 0.971
Epoch 5/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.1552 - acc: 0.9533 - val_loss: 0.0873 - val_acc: 0.973
Epoch 6/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.1393 - acc: 0.9583 - val_loss: 0.0802 - val_acc: 0.9752
Epoch 7/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.1280 - acc: 0.9612 - val_loss: 0.0845 - val_acc: 0.974
Epoch 8/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.1198 - acc: 0.9635 - val_loss: 0.0760 - val_acc: 0.977
Epoch 9/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.1102 - acc: 0.9659 - val_loss: 0.0706 - val_acc: 0.978
Epoch 10/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.1018 - acc: 0.9693 - val_loss: 0.0666 - val_acc: 0.979
Epoch 11/20
60000/60000 [==============================] - 7s 120us/step - loss: 0.0991 - acc: 0.9694 - val_loss: 0.0717 - val_acc: 0.978
Epoch 12/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0935 - acc: 0.9708 - val_loss: 0.0677 - val_acc: 0.979
Epoch 13/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.0880 - acc: 0.9725 - val_loss: 0.0648 - val_acc: 0.980
Epoch 14/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.0845 - acc: 0.9737 - val_loss: 0.0688 - val_acc: 0.979
Epoch 15/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0834 - acc: 0.9739 - val_loss: 0.0652 - val_acc: 0.981
Epoch 16/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.0796 - acc: 0.9744 - val_loss: 0.0606 - val_acc: 0.981
Epoch 17/20
60000/60000 [==============================] - 7s 117us/step - loss: 0.0743 - acc: 0.9780 - val_loss: 0.0646 - val_acc: 0.981
Epoch 18/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0717 - acc: 0.9777 - val_loss: 0.0636 - val_acc: 0.982
Epoch 19/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.0706 - acc: 0.9780 - val_loss: 0.0608 - val_acc: 0.982
Epoch 20/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.0667 - acc: 0.9794 - val_loss: 0.0615 - val_acc: 0.9816
Test score: 0.061509001989616084
Test accuracy: 0.9816
```







Architecture 2 : Layer1(500), Layer2(250), Layer3(125)

i) MLP + ReLU + ADAM

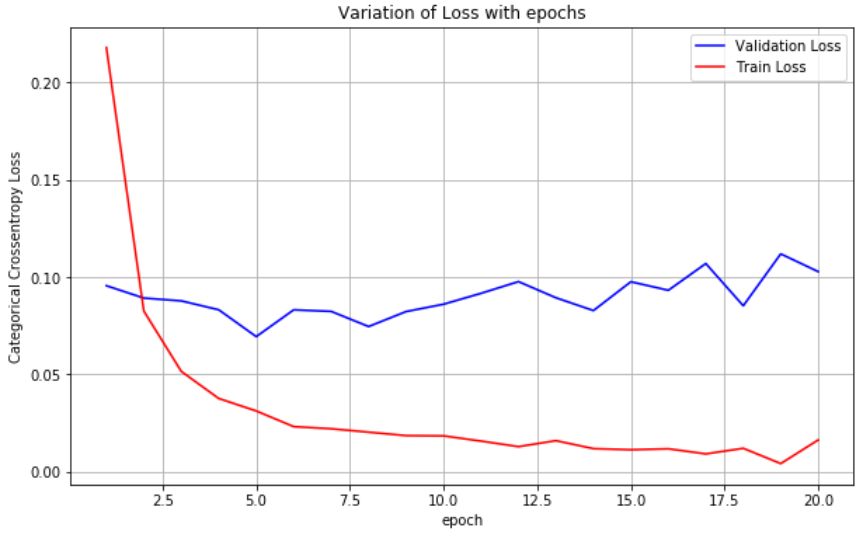
```
import warnings
warnings.filterwarnings("ignore")
```

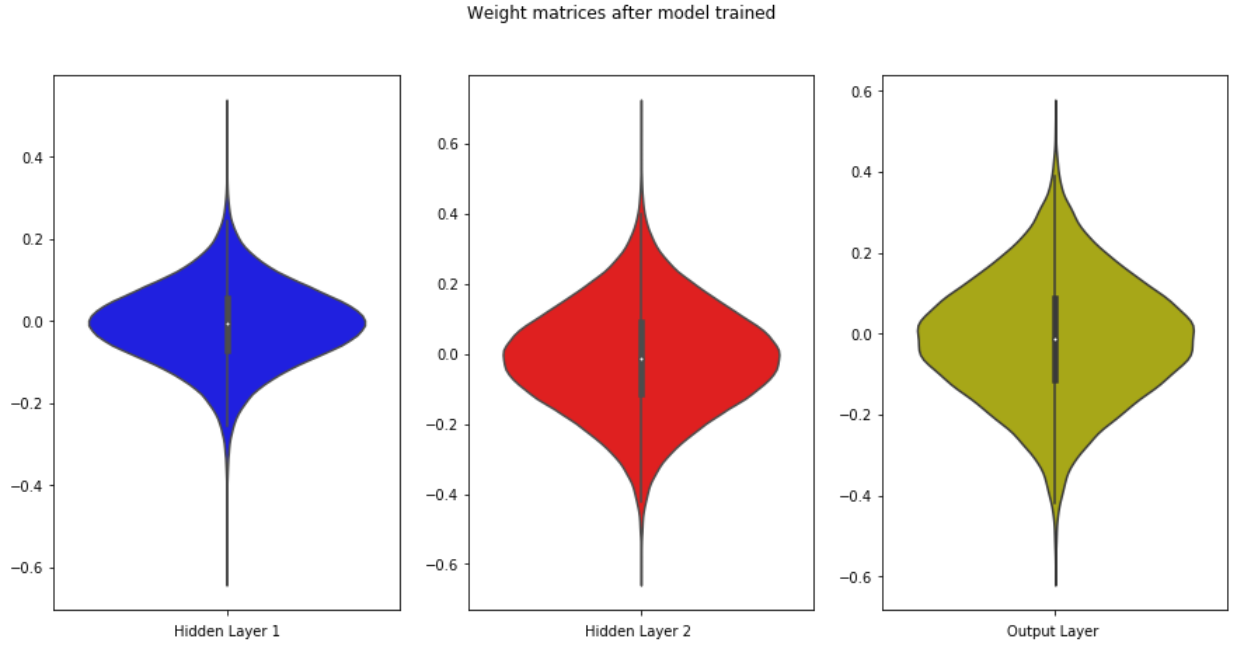
```
model_relu2 = Sequential()
model_relu2.add(Dense(500, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.062, seed=Nor
model_relu2.add(Dense(250, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)))
model_relu2.add(Dense(125, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)))
model_relu2.add(Dense(output_dim, activation='softmax'))

print(model_relu2.summary())

model_relu2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu2.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Model: "sequential_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_51 (Dense)             (None, 500)               392500
_________________________________________________________________
dense_52 (Dense)             (None, 250)               125250
_________________________________________________________________
dense_53 (Dense)             (None, 125)               31375
_________________________________________________________________
dense_54 (Dense)             (None, 10)                1260
=================================================================
Total params: 550,385
Trainable params: 550,385
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.2180 - acc: 0.9334 - val_loss: 0.0955 - val_acc: 0.971
Epoch 2/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0825 - acc: 0.9746 - val_loss: 0.0892 - val_acc: 0.9725
Epoch 3/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0514 - acc: 0.9834 - val_loss: 0.0877 - val_acc: 0.9720
Epoch 4/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0375 - acc: 0.9879 - val_loss: 0.0831 - val_acc: 0.9748
Epoch 5/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0311 - acc: 0.9895 - val_loss: 0.0693 - val_acc: 0.9787
Epoch 6/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0230 - acc: 0.9924 - val_loss: 0.0831 - val_acc: 0.9781
Epoch 7/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0220 - acc: 0.9925 - val_loss: 0.0823 - val_acc: 0.9802
Epoch 8/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0202 - acc: 0.9932 - val_loss: 0.0745 - val_acc: 0.9801
Epoch 9/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0184 - acc: 0.9936 - val_loss: 0.0822 - val_acc: 0.9799
Epoch 10/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0183 - acc: 0.9936 - val_loss: 0.0860 - val_acc: 0.9796
Epoch 11/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0156 - acc: 0.9946 - val_loss: 0.0916 - val_acc: 0.9797
Epoch 12/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0128 - acc: 0.9959 - val_loss: 0.0976 - val_acc: 0.9790
Epoch 13/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0158 - acc: 0.9950 - val_loss: 0.0893 - val_acc: 0.9805
Epoch 14/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0117 - acc: 0.9961 - val_loss: 0.0827 - val_acc: 0.9823
Epoch 15/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0112 - acc: 0.9965 - val_loss: 0.0976 - val_acc: 0.9809
Epoch 16/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0116 - acc: 0.9962 - val_loss: 0.0932 - val_acc: 0.9803
Epoch 17/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0090 - acc: 0.9972 - val_loss: 0.1069 - val_acc: 0.9782
Epoch 18/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0119 - acc: 0.9963 - val_loss: 0.0852 - val_acc: 0.9813
Epoch 19/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0040 - acc: 0.9987 - val_loss: 0.1119 - val_acc: 0.9799
Epoch 20/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0162 - acc: 0.9952 - val_loss: 0.1028 - val_acc: 0.9808
```

```
1 loss_plot(model_relu2)
2 weight_plot(model_relu2)
```

```
Test score: 0.10277052689156026
Test accuracy: 0.9808
```





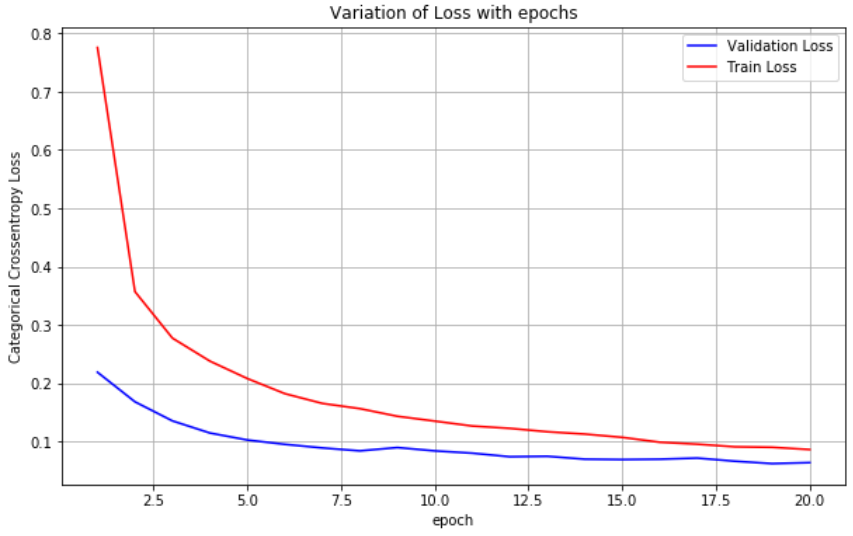
ii) MLP + Batch Normalization + Dropout + AdamOptimizer

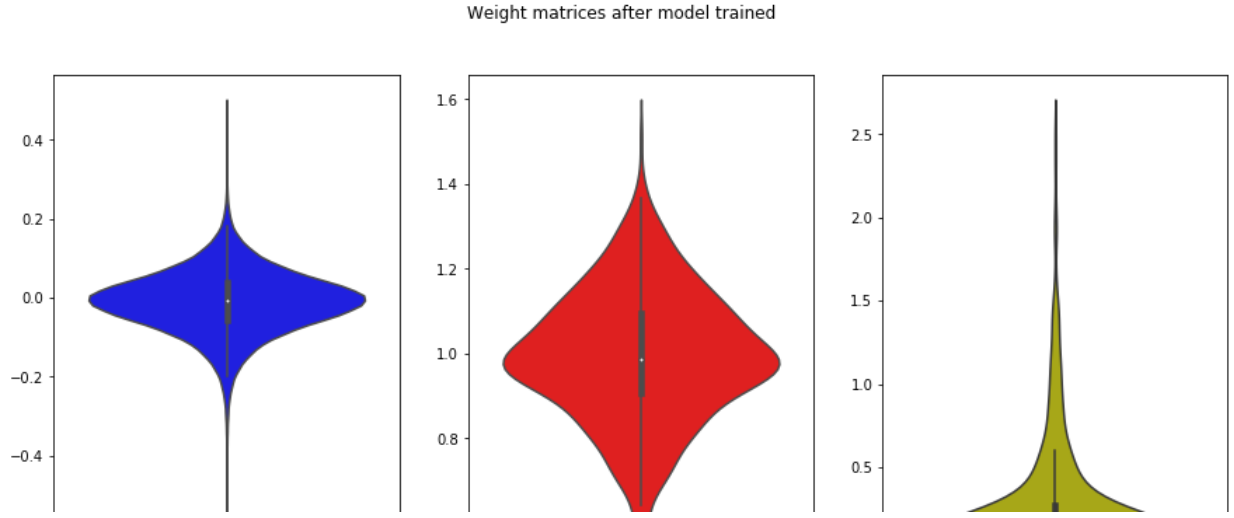
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization-function-in-keras

from keras.layers import Dropout

model_drop2 = Sequential()

model_drop2.add(Dense(500, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=Nor
model_drop2.add(BatchNormalization())
model_drop2.add(Dropout(0.5))

model_drop2.add(Dense(250, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop2.add(BatchNormalization())
model_drop2.add(Dropout(0.5))

model_drop2.add(Dense(125, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop2.add(BatchNormalization())
model_drop2.add(Dropout(0.5))

model_drop2.add(Dense(output_dim, activation='softmax'))


model_drop2.summary()
```

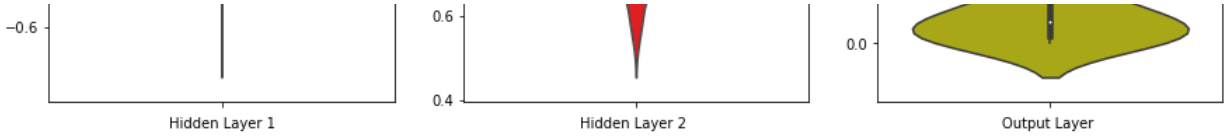
```
Model: "sequential_16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_55 (Dense)             (None, 500)               392500
_________________________________________________________________
batch_normalization_21 (Batc (None, 500)               2000
_________________________________________________________________
dropout_17 (Dropout)         (None, 500)               0
_________________________________________________________________
dense_56 (Dense)             (None, 250)               125250
_________________________________________________________________
batch_normalization_22 (Batc (None, 250)               1000
_________________________________________________________________
dropout_18 (Dropout)         (None, 250)               0
_________________________________________________________________
dense_57 (Dense)             (None, 125)               31375
_________________________________________________________________
batch_normalization_23 (Batc (None, 125)               500
_________________________________________________________________
dropout_19 (Dropout)         (None, 125)               0
_________________________________________________________________
dense_58 (Dense)             (None, 10)                1260
=================================================================
Total params: 553,885
Trainable params: 552,135
Non-trainable params: 1,750
_________________________________________________________________
```

```
model_drop2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_drop2.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
loss_plot(model_drop2)
weight_plot(model_drop2)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 12s 206us/step - loss: 0.7758 - acc: 0.7553 - val_loss: 0.2188 - val_acc: 0.93
Epoch 2/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.3571 - acc: 0.8941 - val_loss: 0.1681 - val_acc: 0.947
Epoch 3/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.2773 - acc: 0.9179 - val_loss: 0.1353 - val_acc: 0.959
Epoch 4/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.2378 - acc: 0.9301 - val_loss: 0.1146 - val_acc: 0.963
Epoch 5/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.2079 - acc: 0.9394 - val_loss: 0.1025 - val_acc: 0.968
Epoch 6/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.1820 - acc: 0.9472 - val_loss: 0.0951 - val_acc: 0.970
Epoch 7/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.1653 - acc: 0.9516 - val_loss: 0.0889 - val_acc: 0.972
Epoch 8/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.1564 - acc: 0.9542 - val_loss: 0.0839 - val_acc: 0.973
Epoch 9/20
60000/60000 [==============================] - 8s 140us/step - loss: 0.1432 - acc: 0.9573 - val_loss: 0.0896 - val_acc: 0.972
Epoch 10/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.1350 - acc: 0.9607 - val_loss: 0.0839 - val_acc: 0.975
Epoch 11/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.1265 - acc: 0.9628 - val_loss: 0.0800 - val_acc: 0.977
Epoch 12/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.1225 - acc: 0.9637 - val_loss: 0.0739 - val_acc: 0.977
Epoch 13/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.1167 - acc: 0.9655 - val_loss: 0.0746 - val_acc: 0.978
Epoch 14/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.1127 - acc: 0.9670 - val_loss: 0.0697 - val_acc: 0.978
Epoch 15/20
60000/60000 [==============================] - 8s 128us/step - loss: 0.1071 - acc: 0.9678 - val_loss: 0.0690 - val_acc: 0.980
Epoch 16/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.0987 - acc: 0.9707 - val_loss: 0.0696 - val_acc: 0.979
Epoch 17/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.0954 - acc: 0.9714 - val_loss: 0.0715 - val_acc: 0.980
Epoch 18/20
60000/60000 [==============================] - 8s 128us/step - loss: 0.0909 - acc: 0.9730 - val_loss: 0.0662 - val_acc: 0.981
Epoch 19/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.0900 - acc: 0.9726 - val_loss: 0.0621 - val_acc: 0.982
Epoch 20/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.0862 - acc: 0.9739 - val_loss: 0.0637 - val_acc: 0.981
Test score: 0.06373635350148543
Test accuracy: 0.9818
```







Architecture 3 : Layer1(700), Layer2(350), Layer3(200), Layer4(100), Layer5(50)

i) MLP + ReLU + ADAM

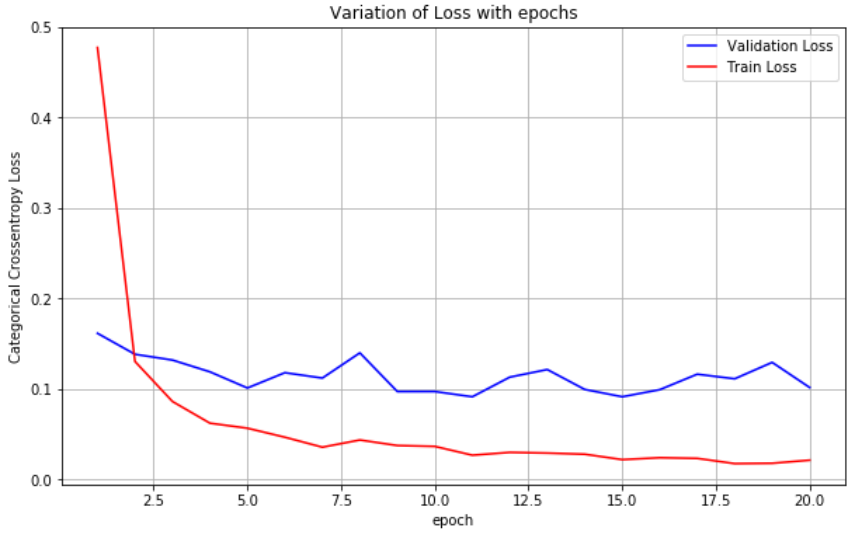
```
import warnings
warnings.filterwarnings("ignore")
```

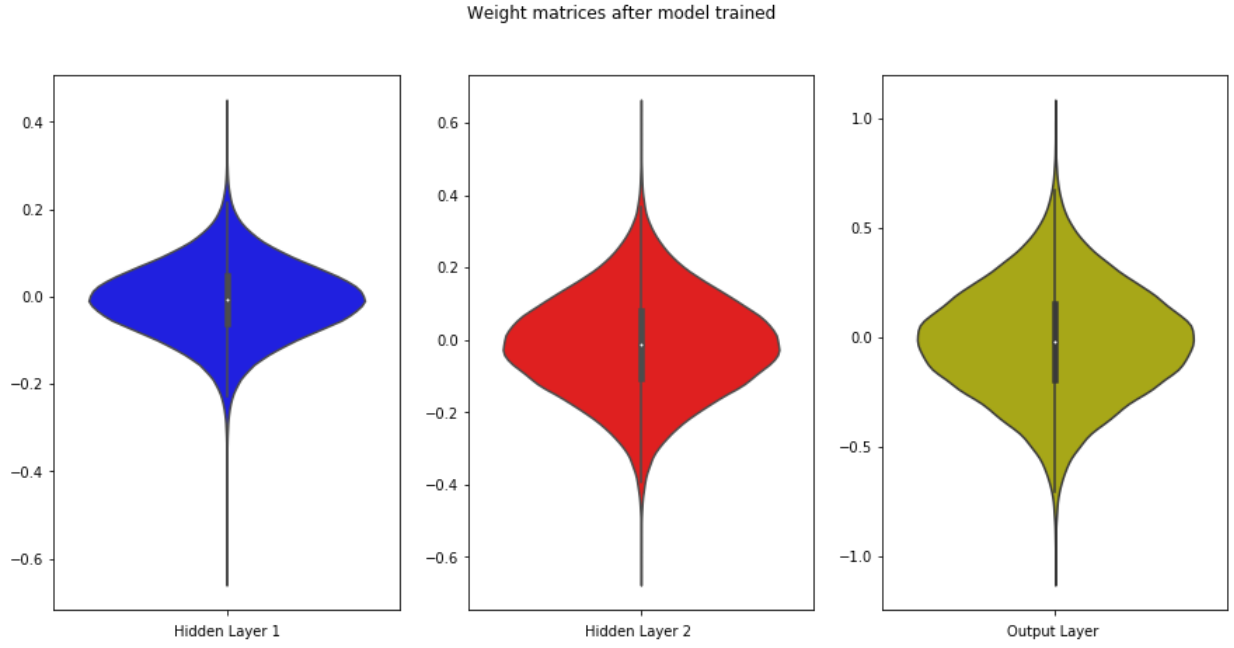
```
model_relu3 = Sequential()
model_relu3.add(Dense(700, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.062, seed=Nor
model_relu3.add(Dense(350, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)))
model_relu3.add(Dense(200, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.25, seed=None)))
model_relu3.add(Dense(100, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.1, seed=None)))
model_relu3.add(Dense(50, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.5, seed=None)))
model_relu3.add(Dense(output_dim, activation='softmax'))

print(model_relu3.summary())

model_relu3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_relu3.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Model: "sequential_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_59 (Dense)             (None, 700)               549500
_________________________________________________________________
dense_60 (Dense)             (None, 350)               245350
_________________________________________________________________
dense_61 (Dense)             (None, 200)               70200
_________________________________________________________________
dense_62 (Dense)             (None, 100)               20100
_________________________________________________________________
dense_63 (Dense)             (None, 50)                5050
_________________________________________________________________
dense_64 (Dense)             (None, 10)                510
=================================================================
Total params: 890,710
Trainable params: 890,710
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.4770 - acc: 0.9003 - val_loss: 0.1612 - val_acc: 0.951
Epoch 2/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.1302 - acc: 0.9619 - val_loss: 0.1381 - val_acc: 0.9604
Epoch 3/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0859 - acc: 0.9739 - val_loss: 0.1317 - val_acc: 0.9617
Epoch 4/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.0619 - acc: 0.9807 - val_loss: 0.1187 - val_acc: 0.9691
Epoch 5/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0564 - acc: 0.9819 - val_loss: 0.1008 - val_acc: 0.9728
Epoch 6/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0464 - acc: 0.9849 - val_loss: 0.1177 - val_acc: 0.9682
Epoch 7/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0354 - acc: 0.9881 - val_loss: 0.1118 - val_acc: 0.9709
Epoch 8/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0433 - acc: 0.9867 - val_loss: 0.1396 - val_acc: 0.9664
Epoch 9/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.0372 - acc: 0.9885 - val_loss: 0.0969 - val_acc: 0.9741
Epoch 10/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0363 - acc: 0.9888 - val_loss: 0.0969 - val_acc: 0.9745
Epoch 11/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0265 - acc: 0.9920 - val_loss: 0.0911 - val_acc: 0.9767
Epoch 12/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0296 - acc: 0.9908 - val_loss: 0.1128 - val_acc: 0.9737
Epoch 13/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.0289 - acc: 0.9911 - val_loss: 0.1212 - val_acc: 0.9730
Epoch 14/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0276 - acc: 0.9913 - val_loss: 0.0992 - val_acc: 0.9749
Epoch 15/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0217 - acc: 0.9933 - val_loss: 0.0911 - val_acc: 0.9794
Epoch 16/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0237 - acc: 0.9930 - val_loss: 0.0989 - val_acc: 0.9794
Epoch 17/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0230 - acc: 0.9936 - val_loss: 0.1160 - val_acc: 0.9764
Epoch 18/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0172 - acc: 0.9947 - val_loss: 0.1110 - val_acc: 0.9766
Epoch 19/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0176 - acc: 0.9949 - val_loss: 0.1291 - val_acc: 0.9745
Epoch 20/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0210 - acc: 0.9939 - val_loss: 0.1013 - val_acc: 0.9776
```

```
1 loss_plot(model_relu3)
2 weight_plot(model_relu3)
```

```
Test score: 0.10133317776405511
Test accuracy: 0.9776
```





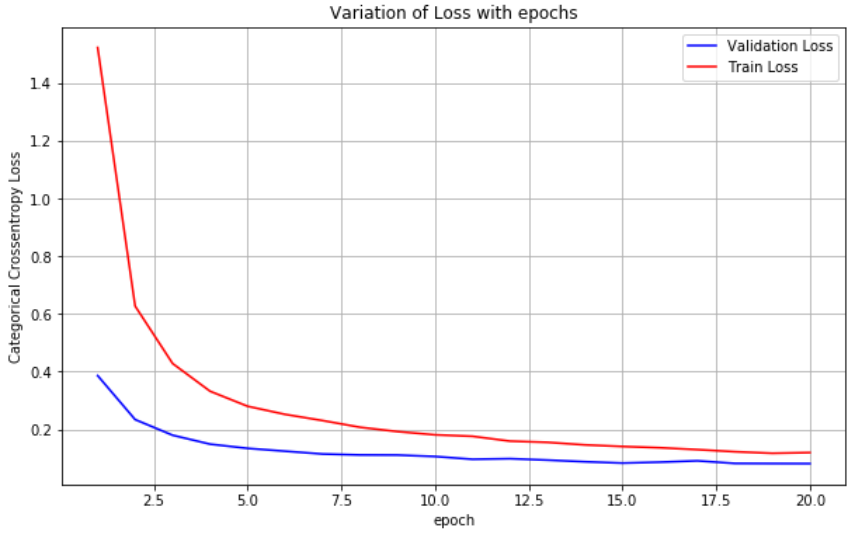
ii) MLP + Batch Normalization + Dropout + AdamOptimizer

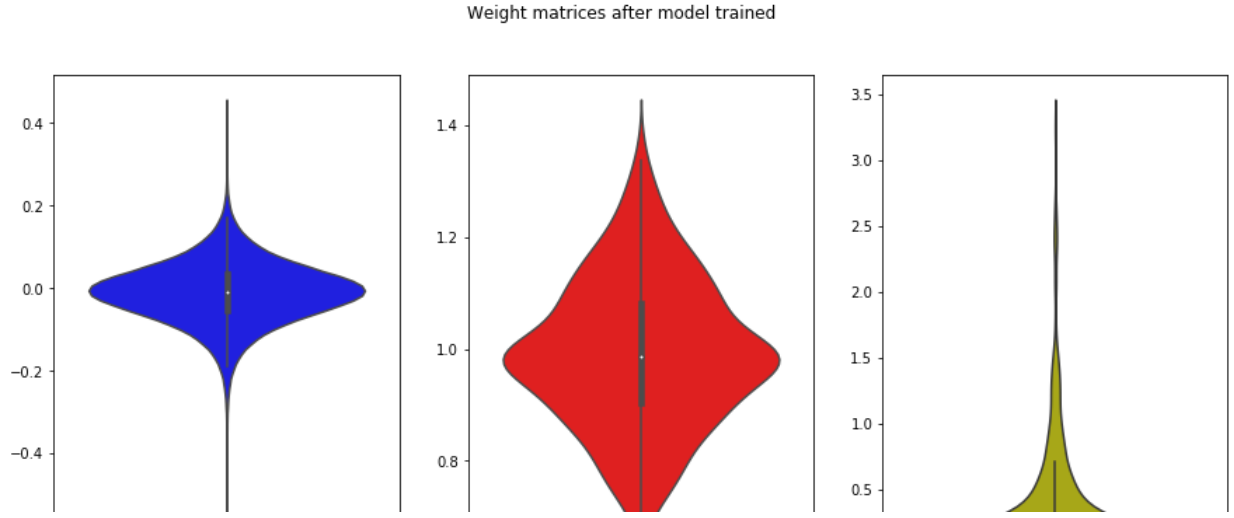
```python
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormalization-function-in-keras

from keras.layers import Dropout

model_drop3 = Sequential()

model_drop3.add(Dense(700, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=Nor
model_drop3.add(BatchNormalization())
model_drop3.add(Dropout(0.5))

model_drop3.add(Dense(350, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.25, seed=None)))
model_drop3.add(BatchNormalization())
model_drop3.add(Dropout(0.5))

model_drop3.add(Dense(200, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)))
model_drop3.add(BatchNormalization())
model_drop3.add(Dropout(0.5))

model_drop3.add(Dense(100, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.15, seed=None)))
model_drop3.add(BatchNormalization())
model_drop3.add(Dropout(0.5))

model_drop3.add(Dense(50, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.3, seed=None)))
model_drop3.add(BatchNormalization())
model_drop3.add(Dropout(0.5))

model_drop3.add(Dense(output_dim, activation='softmax'))


model_drop3.summary()
```

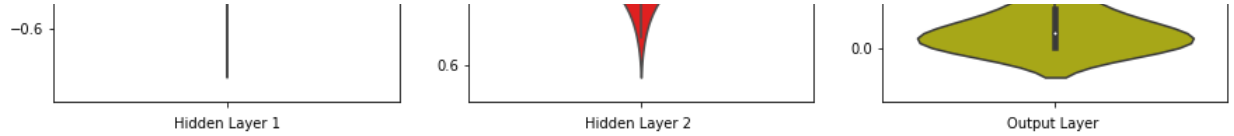
```
Model: "sequential_18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_65 (Dense)             (None, 700)               549500
_________________________________________________________________
batch_normalization_24 (Batc (None, 700)               2800
_________________________________________________________________
dropout_20 (Dropout)         (None, 700)               0
_________________________________________________________________
dense_66 (Dense)             (None, 350)               245350
_________________________________________________________________
batch_normalization_25 (Batc (None, 350)               1400
_________________________________________________________________
dropout_21 (Dropout)         (None, 350)               0
_________________________________________________________________
dense_67 (Dense)             (None, 200)               70200
_________________________________________________________________
batch_normalization_26 (Batc (None, 200)               800
_________________________________________________________________
dropout_22 (Dropout)         (None, 200)               0
_________________________________________________________________
dense_68 (Dense)             (None, 100)               20100
_________________________________________________________________
batch_normalization_27 (Batc (None, 100)               400
_________________________________________________________________
dropout_23 (Dropout)         (None, 100)               0
_________________________________________________________________
dense_69 (Dense)             (None, 50)                5050
_________________________________________________________________
batch_normalization_28 (Batc (None, 50)                200
_________________________________________________________________
dropout_24 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_70 (Dense)             (None, 10)                510
=================================================================
Total params: 896,310
Trainable params: 893,510
Non-trainable params: 2,800
_________________________________________________________________
```

```
model_drop3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model_drop3.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
loss_plot(model_drop3)
weight_plot(model_drop3)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 15s 250us/step - loss: 1.5225 - acc: 0.5005 - val_loss: 0.3861 - val_acc: 0.89
Epoch 2/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.6269 - acc: 0.8081 - val_loss: 0.2339 - val_acc: 0.93
Epoch 3/20
60000/60000 [==============================] - 11s 178us/step - loss: 0.4280 - acc: 0.8796 - val_loss: 0.1796 - val_acc: 0.94
Epoch 4/20
60000/60000 [==============================] - 10s 175us/step - loss: 0.3320 - acc: 0.9116 - val_loss: 0.1487 - val_acc: 0.95
Epoch 5/20
60000/60000 [==============================] - 11s 178us/step - loss: 0.2799 - acc: 0.9259 - val_loss: 0.1342 - val_acc: 0.96
Epoch 6/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.2519 - acc: 0.9355 - val_loss: 0.1243 - val_acc: 0.96
Epoch 7/20
60000/60000 [==============================] - 11s 178us/step - loss: 0.2305 - acc: 0.9401 - val_loss: 0.1146 - val_acc: 0.96
Epoch 8/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.2072 - acc: 0.9464 - val_loss: 0.1113 - val_acc: 0.97
Epoch 9/20
60000/60000 [==============================] - 11s 183us/step - loss: 0.1922 - acc: 0.9512 - val_loss: 0.1109 - val_acc: 0.97
Epoch 10/20
60000/60000 [==============================] - 12s 200us/step - loss: 0.1811 - acc: 0.9542 - val_loss: 0.1060 - val_acc: 0.97
Epoch 11/20
60000/60000 [==============================] - 13s 208us/step - loss: 0.1759 - acc: 0.9556 - val_loss: 0.0963 - val_acc: 0.97
Epoch 12/20
60000/60000 [==============================] - 12s 206us/step - loss: 0.1592 - acc: 0.9609 - val_loss: 0.0984 - val_acc: 0.97
Epoch 13/20
60000/60000 [==============================] - 13s 215us/step - loss: 0.1550 - acc: 0.9612 - val_loss: 0.0935 - val_acc: 0.97
Epoch 14/20
60000/60000 [==============================] - 13s 215us/step - loss: 0.1461 - acc: 0.9639 - val_loss: 0.0877 - val_acc: 0.97
Epoch 15/20
60000/60000 [==============================] - 13s 221us/step - loss: 0.1406 - acc: 0.9645 - val_loss: 0.0832 - val_acc: 0.97
Epoch 16/20
60000/60000 [==============================] - 13s 219us/step - loss: 0.1363 - acc: 0.9666 - val_loss: 0.0864 - val_acc: 0.97
Epoch 17/20
60000/60000 [==============================] - 14s 225us/step - loss: 0.1298 - acc: 0.9680 - val_loss: 0.0910 - val_acc: 0.97
Epoch 18/20
60000/60000 [==============================] - 13s 222us/step - loss: 0.1224 - acc: 0.9698 - val_loss: 0.0820 - val_acc: 0.98
Epoch 19/20
60000/60000 [==============================] - 13s 217us/step - loss: 0.1172 - acc: 0.9707 - val_loss: 0.0816 - val_acc: 0.98
Epoch 20/20
60000/60000 [==============================] - 13s 212us/step - loss: 0.1195 - acc: 0.9709 - val_loss: 0.0813 - val_acc: 0.98
Test score: 0.08133343692359049
Test accuracy: 0.9811
```







Observations:

1. Batch Normalization and Dropout layers reduce the variance of the model(prevent it from overfitting to the train data). The performance of the MLP on train data is much better with BN and Dropout.
2. All the models are able to minimize the loss and give close to 98% accuracy after just 15 epochs.
3. For MNIST data, evan a shallow NN works well, there's not much difference in the performance of models on increasing the depth of the neural network.