MLP on MNIST Dataset

In [1]:

```
import keras
from keras.utils import np_utils
from keras.datasets import mnist
import seaborn as sns
from keras.initializers import RandomNormal
import matplotlib.pyplot as plt
import numpy as np
import time
```

```
Using TensorFlow backend.
```

[1] Dataset Loading and pre-processing

In [2]:

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

In [3]:

```
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)
```

In [4]:

```
# 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 [5]:

```
# 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)
```

In [6]:

```
# 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   0 139 253 190   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 [7]:

```python
# Normalization
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
scaler.fit(X_test)
X_test = scaler.transform(X_test)
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 [8]:

```
# 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.]
```

[2] MLP Models

In [9]:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import Dropout
from keras.layers.normalization import BatchNormalization
```

In [10]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 30
```

In [11]:

```
def plt_epoch_vs_loss(x, vy, ty):
    fig = plt.figure(figsize=(9,7))
```

```
    fig = plt.figure(figsize=(9,7))
    sns.set_style("whitegrid",{'axes.grid' : True})
    plt.plot(x, vy, 'b', label="Validation Loss")
    plt.plot(x, ty, 'r', label="Train Loss")
    plt.legend()
    plt.xlabel("epoch")
    plt.ylabel("Categorical Crossentropy Loss")
    plt.title("Loss")
    plt.show()
```

[2.1] Model 1

Input(784) - ReLu(BatchNormalization(256)) - Dropout(0.5) - ReLu(BatchNormalization(128)) - Dropout(0.25) - Softmax(Output(10)) - Adam Optimizer

In [12]:

```
# for relu layers we are using 'He-Normal weight Initialization'
# If we sample weights from a normal distribution N(0,σ) we satisfy this condition with
σ=√(2/(fan_in).
# h1 =>  σ=√(2/(256) = 0.088  => N(0,σ) = N(0,0.088)
# h2 =>  σ=√(2/(128) = 0.125  => N(0,σ) = N(0,0.125)

model1 = Sequential()
model1.add(Dense(256, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(
mean=0.0, stddev=0.088, seed=None)))
model1.add(BatchNormalization())
model1.add(Dropout(0.5))

model1.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, se
ed=None)) )
model1.add(BatchNormalization())
model1.add(Dropout(0.25))

model1.add(Dense(output_dim, activation='softmax'))

model1.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 256)               200960    
_________________________________________________________________
batch_normalization_1 (Batch (None, 256)               1024      
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 128)               32896     
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512       
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290      
=================================================================
Total params: 236,682
Trainable params: 235,914
Non-trainable params: 768
_________________________________________________________________
```

In [13]:

```
model1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model1.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validatio
n_data=(X_test, Y_test))
```

```
WARNING:tensorflow:From C:\Users\sanjeev\Anaconda3\lib\site-
packages\keras\backend\tensorflow_backend.py:422: The name tf.global_variables is deprecated. Plea
se use tf.compat.v1.global_variables instead.
```

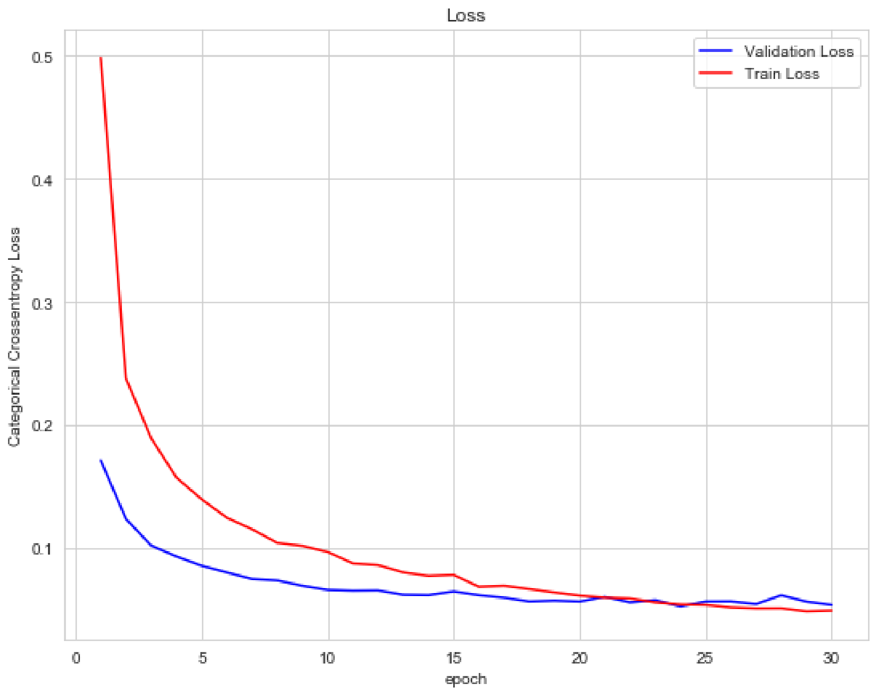
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 6s 97us/step - loss: 0.4985 - accuracy: 0.8469 - va
l_loss: 0.1711 - val_accuracy: 0.9486
Epoch 2/30
60000/60000 [==============================] - 4s 73us/step - loss: 0.2377 - accuracy: 0.9275 - va
l_loss: 0.1235 - val_accuracy: 0.9611
Epoch 3/30
60000/60000 [==============================] - 4s 70us/step - loss: 0.1889 - accuracy: 0.9424 - va
l_loss: 0.1017 - val_accuracy: 0.9688
Epoch 4/30
60000/60000 [==============================] - 4s 64us/step - loss: 0.1571 - accuracy: 0.9520 - va
l_loss: 0.0929 - val_accuracy: 0.9706
Epoch 5/30
60000/60000 [==============================] - 4s 66us/step - loss: 0.1395 - accuracy: 0.9567 - va
l_loss: 0.0855 - val_accuracy: 0.9735
Epoch 6/30
60000/60000 [==============================] - 5s 78us/step - loss: 0.1244 - accuracy: 0.9609 - va
l_loss: 0.0800 - val_accuracy: 0.9733
Epoch 7/30
60000/60000 [==============================] - 5s 78us/step - loss: 0.1151 - accuracy: 0.9639 - va
l_loss: 0.0747 - val_accuracy: 0.9754
Epoch 8/30
60000/60000 [==============================] - 4s 72us/step - loss: 0.1040 - accuracy: 0.9672 - va
l_loss: 0.0735 - val_accuracy: 0.9774
Epoch 9/30
60000/60000 [==============================] - 4s 67us/step - loss: 0.1015 - accuracy: 0.9683 - va
l_loss: 0.0691 - val_accuracy: 0.9776
Epoch 10/30
60000/60000 [==============================] - 4s 67us/step - loss: 0.0966 - accuracy: 0.9700 - va
l_loss: 0.0658 - val_accuracy: 0.9795
Epoch 11/30
60000/60000 [==============================] - 4s 74us/step - loss: 0.0873 - accuracy: 0.9722 - va
l_loss: 0.0651 - val_accuracy: 0.9791
Epoch 12/30
60000/60000 [==============================] - 4s 72us/step - loss: 0.0860 - accuracy: 0.9725 - va
l_loss: 0.0653 - val_accuracy: 0.9794
Epoch 13/30
60000/60000 [==============================] - 4s 70us/step - loss: 0.0801 - accuracy: 0.9743 - va
l_loss: 0.0618 - val_accuracy: 0.9796
Epoch 14/30
60000/60000 [==============================] - 4s 70us/step - loss: 0.0772 - accuracy: 0.9756 - va
l_loss: 0.0616 - val_accuracy: 0.9809
Epoch 15/30
60000/60000 [==============================] - 4s 63us/step - loss: 0.0780 - accuracy: 0.9750 - va
l_loss: 0.0645 - val_accuracy: 0.9795
Epoch 16/30
60000/60000 [==============================] - 4s 63us/step - loss: 0.0683 - accuracy: 0.9776 - va
l_loss: 0.0615 - val_accuracy: 0.9803
Epoch 17/30
60000/60000 [==============================] - 4s 64us/step - loss: 0.0691 - accuracy: 0.9776 - va
l_loss: 0.0594 - val_accuracy: 0.9822
Epoch 18/30
60000/60000 [==============================] - 4s 65us/step - loss: 0.0665 - accuracy: 0.9780 - va
l_loss: 0.0563 - val_accuracy: 0.9819
Epoch 19/30
60000/60000 [==============================] - 4s 64us/step - loss: 0.0637 - accuracy: 0.9795 - va
l_loss: 0.0570 - val_accuracy: 0.9826
Epoch 20/30
60000/60000 [==============================] - 4s 64us/step - loss: 0.0612 - accuracy: 0.9799 - va
l_loss: 0.0563 - val_accuracy: 0.9827
Epoch 21/30
60000/60000 [==============================] - 4s 66us/step - loss: 0.0595 - accuracy: 0.9801 - va
l_loss: 0.0601 - val_accuracy: 0.9815
Epoch 22/30
60000/60000 [==============================] - 4s 67us/step - loss: 0.0588 - accuracy: 0.9810 - va
l_loss: 0.0556 - val_accuracy: 0.9823
Epoch 23/30
60000/60000 [==============================] - 5s 75us/step - loss: 0.0556 - accuracy: 0.9820 - va
l_loss: 0.0574 - val_accuracy: 0.9820
Epoch 24/30
60000/60000 [==============================] - 5s 81us/step - loss: 0.0540 - accuracy: 0.9829 - va
l_loss: 0.0523 - val_accuracy: 0.9836
Epoch 25/30
60000/60000 [==============================] - 4s 74us/step - loss: 0.0536 - accuracy: 0.9823 - va
l_loss: 0.0563 - val_accuracy: 0.9834
Epoch 26/30
```

```
60000/60000 [==============================] - 4s 63us/step - loss: 0.0515 - accuracy: 0.9831 - va
l_loss: 0.0564 - val_accuracy: 0.9833
Epoch 27/30
60000/60000 [==============================] - 4s 65us/step - loss: 0.0507 - accuracy: 0.9829 - va
l_loss: 0.0543 - val_accuracy: 0.9836
Epoch 28/30
60000/60000 [==============================] - 4s 64us/step - loss: 0.0508 - accuracy: 0.9835 - va
l_loss: 0.0614 - val_accuracy: 0.9828
Epoch 29/30
60000/60000 [==============================] - 4s 67us/step - loss: 0.0483 - accuracy: 0.9842 - va
l_loss: 0.0562 - val_accuracy: 0.9826
Epoch 30/30
60000/60000 [==============================] - 4s 72us/step - loss: 0.0489 - accuracy: 0.9836 - va
l_loss: 0.0536 - val_accuracy: 0.9841
```

In [14]:

```python
score1 = model1.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score1[0])
print('Test accuracy:', score1[1])

# list of epoch numbers
epochs = list(range(1,nb_epoch+1))
val_loss = history.history['val_loss']
train_loss = history.history['loss']
plt_epoch_vs_loss(epochs, val_loss, train_loss)
```

```
Test score: 0.05358278493771504
Test accuracy: 0.9840999841690063
```



In [15]:

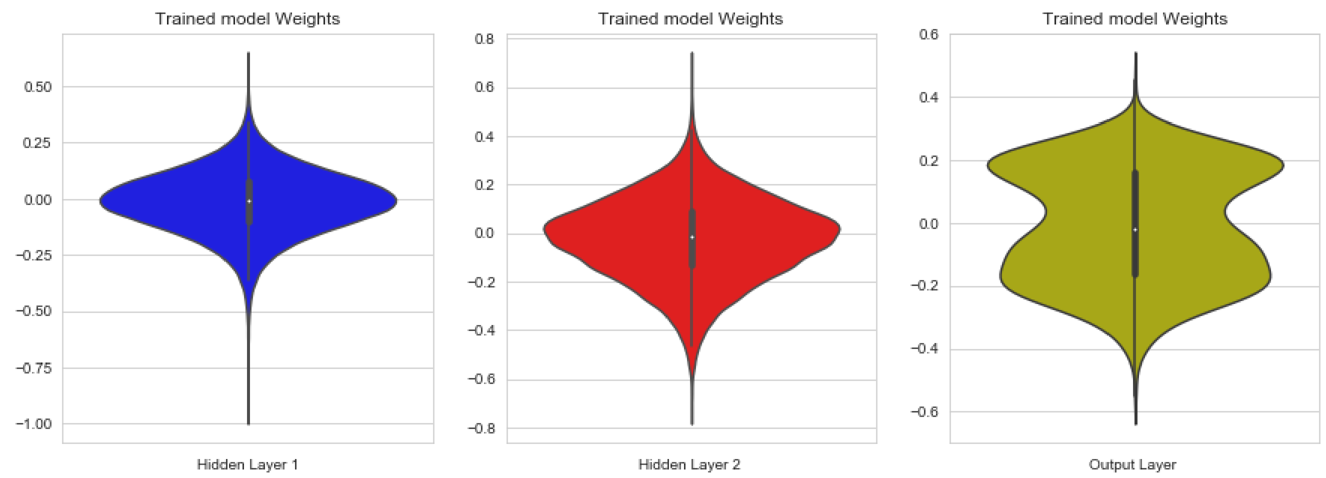
```python
w_after = model1.get_weights()
print(len(w_after))
```

```
14
```

In [16]:

```python
h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
out_w = w_after[12].flatten().reshape(-1,1)


fig = plt.figure(figsize=(15,5))
```

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

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

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



[2.2] Model2

Input(784) - ReLu(BatchNormalization(512)) - Dropout(0.5) - ReLu(BatchNormalization(256)) - Dropout(0.25) - ReLu(BatchNormalization(128)) - Dropout(0.125) - Softmax(Output(10)) - Adam Optimizer

In [17]:

```
# for relu layers we are using 'Xavier/Glorot-Normal weight Initialization'
# If we sample weights from a normal distribution N(0,σ) we satisfy this condition with
σ=2/(fan_in+fan_out).

from keras.initializers import glorot_normal

model2 = Sequential()

model2.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=glorot_normal
(seed=None)))
model2.add(BatchNormalization())
model2.add(Dropout(0.5))

model2.add(Dense(256, activation='relu', kernel_initializer=glorot_normal(seed=None)))
model2.add(BatchNormalization())
model2.add(Dropout(0.25))

model2.add(Dense(128, activation='relu', kernel_initializer=glorot_normal(seed=None)) )
model2.add(BatchNormalization())
model2.add(Dropout(0.125))

model2.add(Dense(output_dim, activation='softmax'))

model2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_4 (Dense)              (None, 512)               401920    
```

```
dense_4 (Dense)              (None, 512)               401920
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 256)               131328
_________________________________________________________________
batch_normalization_4 (Batch (None, 256)               1024
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               32896
_________________________________________________________________
batch_normalization_5 (Batch (None, 128)               512
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 571,018
Trainable params: 569,226
Non-trainable params: 1,792
_________________________________________________________________
```

In [18]:

```
model2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model2.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validatio
n_data=(X_test, Y_test))
```

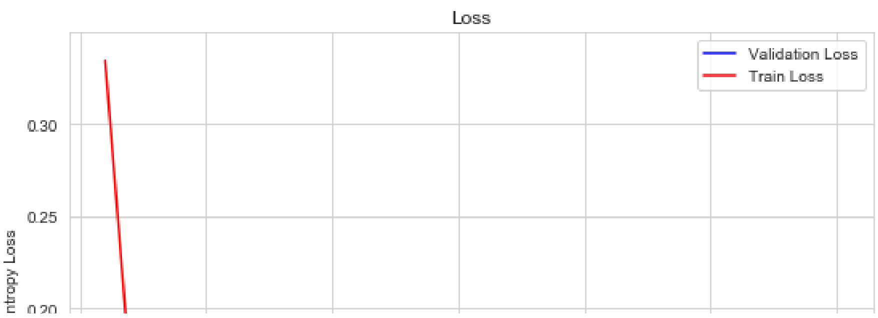
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 10s 169us/step - loss: 0.3345 - accuracy: 0.8966 -
val_loss: 0.1244 - val_accuracy: 0.9605
Epoch 2/30
60000/60000 [==============================] - 9s 146us/step - loss: 0.1651 - accuracy: 0.9495 - v
al_loss: 0.0937 - val_accuracy: 0.9716
Epoch 3/30
60000/60000 [==============================] - 9s 150us/step - loss: 0.1328 - accuracy: 0.9582 - v
al_loss: 0.0817 - val_accuracy: 0.9732
Epoch 4/30
60000/60000 [==============================] - 9s 142us/step - loss: 0.1169 - accuracy: 0.9642 - v
al_loss: 0.0764 - val_accuracy: 0.9767
Epoch 5/30
60000/60000 [==============================] - 9s 145us/step - loss: 0.1008 - accuracy: 0.9683 - v
al_loss: 0.0726 - val_accuracy: 0.9774
Epoch 6/30
60000/60000 [==============================] - 9s 144us/step - loss: 0.0927 - accuracy: 0.9708 - v
al_loss: 0.0662 - val_accuracy: 0.9799
Epoch 7/30
60000/60000 [==============================] - 9s 146us/step - loss: 0.0843 - accuracy: 0.9735 - v
al_loss: 0.0671 - val_accuracy: 0.9800
Epoch 8/30
60000/60000 [==============================] - 9s 156us/step - loss: 0.0794 - accuracy: 0.9747 - v
al_loss: 0.0622 - val_accuracy: 0.9807
Epoch 9/30
60000/60000 [==============================] - 9s 151us/step - loss: 0.0727 - accuracy: 0.9766 - v
al_loss: 0.0647 - val_accuracy: 0.9787
Epoch 10/30
60000/60000 [==============================] - 10s 162us/step - loss: 0.0720 - accuracy: 0.9772 -
val_loss: 0.0630 - val_accuracy: 0.9805
Epoch 11/30
60000/60000 [==============================] - 9s 154us/step - loss: 0.0664 - accuracy: 0.9790 - v
al_loss: 0.0557 - val_accuracy: 0.9833
Epoch 12/30
60000/60000 [==============================] - 10s 166us/step - loss: 0.0632 - accuracy: 0.9798 -
val_loss: 0.0602 - val_accuracy: 0.9823
Epoch 13/30
60000/60000 [==============================] - 9s 153us/step - loss: 0.0605 - accuracy: 0.9800 - v
al_loss: 0.0543 - val_accuracy: 0.9846
Epoch 14/30
60000/60000 [==============================] - 9s 151us/step - loss: 0.0598 - accuracy: 0.9801 - v
al_loss: 0.0593 - val_accuracy: 0.9825
Epoch 15/30
```

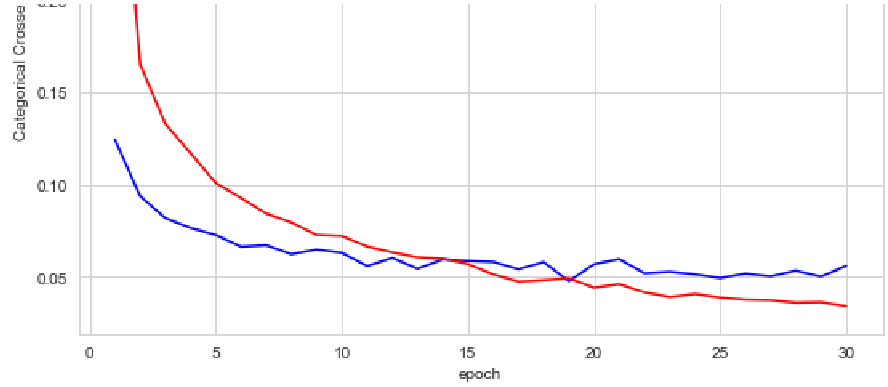
```
60000/60000 [==============================] - 9s 151us/step - loss: 0.0568 - accuracy: 0.9813 - v
al_loss: 0.0586 - val_accuracy: 0.9816
Epoch 16/30
60000/60000 [==============================] - 9s 155us/step - loss: 0.0512 - accuracy: 0.9835 - v
al_loss: 0.0580 - val_accuracy: 0.9840
Epoch 17/30
60000/60000 [==============================] - 9s 156us/step - loss: 0.0473 - accuracy: 0.9845 - v
al_loss: 0.0540 - val_accuracy: 0.9855
Epoch 18/30
60000/60000 [==============================] - 9s 156us/step - loss: 0.0481 - accuracy: 0.9847 - v
al_loss: 0.0579 - val_accuracy: 0.9826
Epoch 19/30
60000/60000 [==============================] - 9s 148us/step - loss: 0.0491 - accuracy: 0.9838 - v
al_loss: 0.0476 - val_accuracy: 0.9851
Epoch 20/30
60000/60000 [==============================] - 9s 145us/step - loss: 0.0440 - accuracy: 0.9856 - v
al_loss: 0.0567 - val_accuracy: 0.9845
Epoch 21/30
60000/60000 [==============================] - 9s 150us/step - loss: 0.0460 - accuracy: 0.9852 - v
al_loss: 0.0595 - val_accuracy: 0.9848
Epoch 22/30
60000/60000 [==============================] - 9s 152us/step - loss: 0.0415 - accuracy: 0.9862 - v
al_loss: 0.0519 - val_accuracy: 0.9848
Epoch 23/30
60000/60000 [==============================] - 10s 161us/step - loss: 0.0390 - accuracy: 0.9871 -
val_loss: 0.0527 - val_accuracy: 0.9863
Epoch 24/30
60000/60000 [==============================] - 9s 146us/step - loss: 0.0406 - accuracy: 0.9872 - v
al_loss: 0.0513 - val_accuracy: 0.9850
Epoch 25/30
60000/60000 [==============================] - 9s 144us/step - loss: 0.0387 - accuracy: 0.9873 - v
al_loss: 0.0493 - val_accuracy: 0.9862
Epoch 26/30
60000/60000 [==============================] - 9s 146us/step - loss: 0.0375 - accuracy: 0.9875 - v
al_loss: 0.0517 - val_accuracy: 0.9857
Epoch 27/30
60000/60000 [==============================] - 9s 144us/step - loss: 0.0372 - accuracy: 0.9873 - v
al_loss: 0.0502 - val_accuracy: 0.9861
Epoch 28/30
60000/60000 [==============================] - 9s 144us/step - loss: 0.0359 - accuracy: 0.9884 - v
al_loss: 0.0533 - val_accuracy: 0.9853
Epoch 29/30
60000/60000 [==============================] - 9s 145us/step - loss: 0.0361 - accuracy: 0.9880 - v
al_loss: 0.0500 - val_accuracy: 0.9860
Epoch 30/30
60000/60000 [==============================] - 9s 145us/step - loss: 0.0341 - accuracy: 0.9885 - v
al_loss: 0.0558 - val_accuracy: 0.9841
```

In [19]:

```python
score2 = model2.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score2[0])
print('Test accuracy:', score2[1])

# list of epoch numbers
epochs = list(range(1,nb_epoch+1))
val_loss = history.history['val_loss']
train_loss = history.history['loss']
plt_epoch_vs_loss(epochs, val_loss, train_loss)
```

```
Test score: 0.0558483196992951
Test accuracy: 0.9840999841690063
```





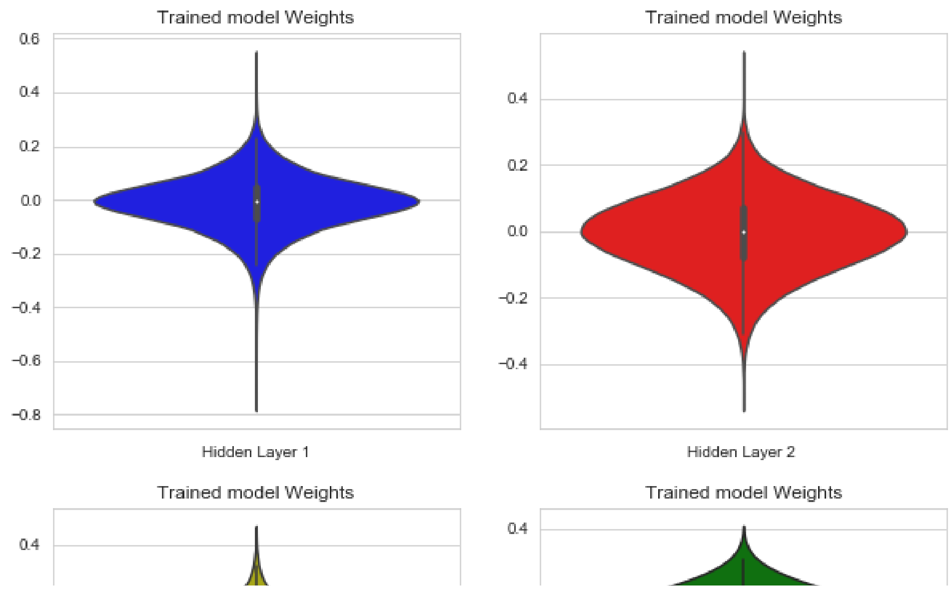
In [20]:

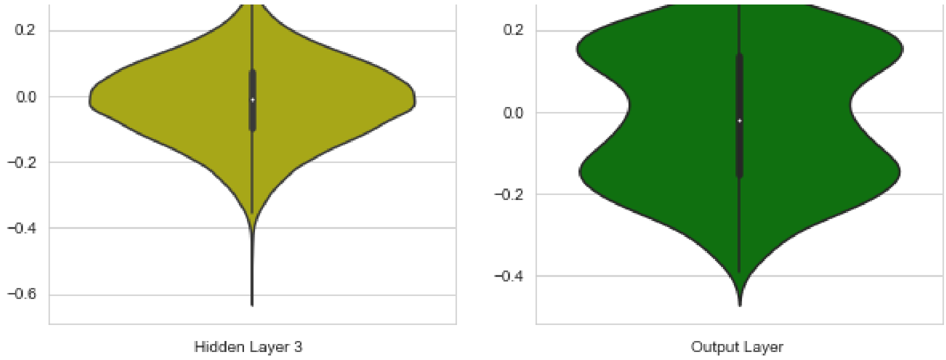
```
w_after = model2.get_weights()
print(len(w_after))
```

```
20
```

In [21]:

```
h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
h3_w = w_after[12].flatten().reshape(-1,1)
out_w = w_after[18].flatten().reshape(-1,1)

fig = plt.figure(figsize=(10,10))
plt.title("Weight matrices after model trained")
plt.subplot(2, 2, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

plt.subplot(2, 2, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w,color='y')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(2, 2, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='g')
plt.xlabel('Output Layer ')
plt.show()
```





[2.3] Model3

Input(784) - ReLu(BatchNormalization(512)) - Dropout(0.5) - ReLu(BatchNormalization(256)) - Dropout(0.4) - ReLu(BatchNormalization(128)) - Dropout(0.3) - ReLu(BatchNormalization(64)) - Dropout(0.2) - ReLu(BatchNormalization(32)) - Dropout(0.1) - Softmax(Output(10)) - Adam Optimizer

In [22]:

```
# for relu layers we are using 'He-Normal weight Initialization'
# If we sample weights from a normal distribution N(0,σ) we satisfy this condition with
σ=2/(fan_in+fan_out).

from keras.initializers import he_normal

model3 = Sequential()

model3.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=he_normal(see
d=None)))
model3.add(BatchNormalization())
model3.add(Dropout(0.5))

model3.add(Dense(256, activation='relu', kernel_initializer=he_normal(seed=None)))
model3.add(BatchNormalization())
model3.add(Dropout(0.4))

model3.add(Dense(128, activation='relu', kernel_initializer=he_normal(seed=None)) )
model3.add(BatchNormalization())
model3.add(Dropout(0.3))

model3.add(Dense(64, activation='relu', kernel_initializer=he_normal(seed=None)) )
model3.add(BatchNormalization())
model3.add(Dropout(0.2))

model3.add(Dense(32, activation='relu', kernel_initializer=he_normal(seed=None)) )
model3.add(BatchNormalization())
model3.add(Dropout(0.1))

model3.add(Dense(output_dim, activation='softmax'))

model3.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 512)               401920
_________________________________________________________________
batch_normalization_6 (Batch (None, 512)               2048
_________________________________________________________________
dropout_6 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_9 (Dense)              (None, 256)               131328
_________________________________________________________________
batch_normalization_7 (Batch (None, 256)               1024
_________________________________________________________________
dropout_7 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_10 (Dense)             (None, 128)               32896
```

```
_________________________________________________________________
batch_normalization_8 (Batch (None, 128)               512       
_________________________________________________________________
dropout_8 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 64)                8256      
_________________________________________________________________
batch_normalization_9 (Batch (None, 64)                256       
_________________________________________________________________
dropout_9 (Dropout)          (None, 64)                0         
_________________________________________________________________
dense_12 (Dense)             (None, 32)                2080      
_________________________________________________________________
batch_normalization_10 (Batc (None, 32)                128       
_________________________________________________________________
dropout_10 (Dropout)         (None, 32)                0         
_________________________________________________________________
dense_13 (Dense)             (None, 10)                330       
=================================================================
Total params: 580,778
Trainable params: 578,794
Non-trainable params: 1,984
_________________________________________________________________
```

In [23]:

```
model3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model3.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

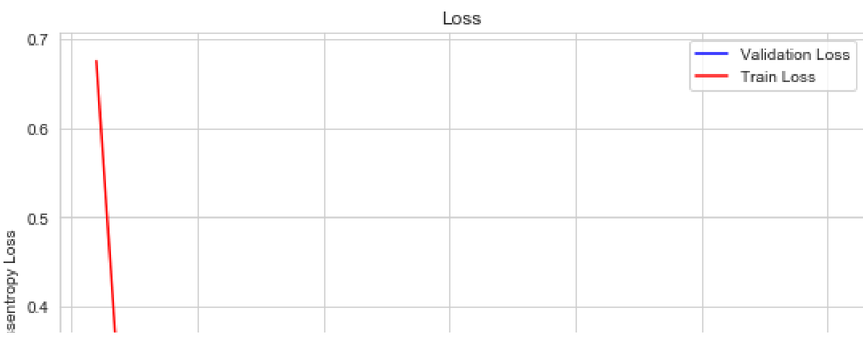
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 12s 194us/step - loss: 0.6755 - accuracy: 0.7921 - val_loss: 0.1842 - val_accuracy: 0.9442
Epoch 2/30
60000/60000 [==============================] - 10s 168us/step - loss: 0.2663 - accuracy: 0.9239 - val_loss: 0.1292 - val_accuracy: 0.9639
Epoch 3/30
60000/60000 [==============================] - 10s 166us/step - loss: 0.2076 - accuracy: 0.9409 - val_loss: 0.1088 - val_accuracy: 0.9686
Epoch 4/30
60000/60000 [==============================] - 10s 167us/step - loss: 0.1793 - accuracy: 0.9498 - val_loss: 0.0968 - val_accuracy: 0.9723
Epoch 5/30
60000/60000 [==============================] - 11s 175us/step - loss: 0.1575 - accuracy: 0.9552 - val_loss: 0.0918 - val_accuracy: 0.9719
Epoch 6/30
60000/60000 [==============================] - 10s 167us/step - loss: 0.1455 - accuracy: 0.9595 - val_loss: 0.0854 - val_accuracy: 0.9745
Epoch 7/30
60000/60000 [==============================] - 11s 182us/step - loss: 0.1315 - accuracy: 0.9624 - val_loss: 0.0867 - val_accuracy: 0.9758
Epoch 8/30
60000/60000 [==============================] - 12s 203us/step - loss: 0.1258 - accuracy: 0.9642 - val_loss: 0.0778 - val_accuracy: 0.9780
Epoch 9/30
60000/60000 [==============================] - 10s 162us/step - loss: 0.1154 - accuracy: 0.9670 - val_loss: 0.0777 - val_accuracy: 0.9770
Epoch 10/30
60000/60000 [==============================] - 10s 165us/step - loss: 0.1065 - accuracy: 0.9694 - val_loss: 0.0750 - val_accuracy: 0.9790
Epoch 11/30
60000/60000 [==============================] - 11s 191us/step - loss: 0.1050 - accuracy: 0.9700 - val_loss: 0.0702 - val_accuracy: 0.9806
Epoch 12/30
60000/60000 [==============================] - 12s 201us/step - loss: 0.0978 - accuracy: 0.9717 - val_loss: 0.0694 - val_accuracy: 0.9804
Epoch 13/30
60000/60000 [==============================] - 12s 195us/step - loss: 0.0936 - accuracy: 0.9726 - val_loss: 0.0739 - val_accuracy: 0.9802
Epoch 14/30
60000/60000 [==============================] - 11s 191us/step - loss: 0.0907 - accuracy: 0.9739 - val_loss: 0.0628 - val_accuracy: 0.9823
Epoch 15/30
60000/60000 [==============================] - 11s 184us/step - loss: 0.0850 - accuracy: 0.9754 -
```

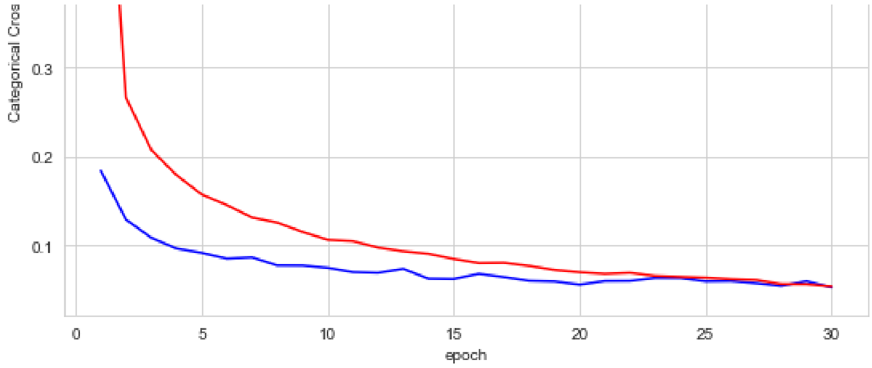
```
60000/60000 [==============================] - 11s 184us/step - loss: 0.0850 - accuracy: 0.9754 -
val_loss: 0.0624 - val_accuracy: 0.9807
Epoch 16/30
60000/60000 [==============================] - 11s 187us/step - loss: 0.0804 - accuracy: 0.9764 -
val_loss: 0.0683 - val_accuracy: 0.9807
Epoch 17/30
60000/60000 [==============================] - 11s 185us/step - loss: 0.0807 - accuracy: 0.9772 -
val_loss: 0.0645 - val_accuracy: 0.9812
Epoch 18/30
60000/60000 [==============================] - 11s 176us/step - loss: 0.0772 - accuracy: 0.9767 -
val_loss: 0.0604 - val_accuracy: 0.9834
Epoch 19/30
60000/60000 [==============================] - 10s 174us/step - loss: 0.0726 - accuracy: 0.9789 -
val_loss: 0.0597 - val_accuracy: 0.9830
Epoch 20/30
60000/60000 [==============================] - 11s 177us/step - loss: 0.0701 - accuracy: 0.9797 -
val_loss: 0.0562 - val_accuracy: 0.9842
Epoch 21/30
60000/60000 [==============================] - 10s 167us/step - loss: 0.0683 - accuracy: 0.9807 -
val_loss: 0.0602 - val_accuracy: 0.9833
Epoch 22/30
60000/60000 [==============================] - 10s 164us/step - loss: 0.0695 - accuracy: 0.9800 -
val_loss: 0.0603 - val_accuracy: 0.9831
Epoch 23/30
60000/60000 [==============================] - 10s 174us/step - loss: 0.0659 - accuracy: 0.9807 -
val_loss: 0.0636 - val_accuracy: 0.9825
Epoch 24/30
60000/60000 [==============================] - 11s 184us/step - loss: 0.0648 - accuracy: 0.9810 -
val_loss: 0.0634 - val_accuracy: 0.9832
Epoch 25/30
60000/60000 [==============================] - 12s 202us/step - loss: 0.0639 - accuracy: 0.9810 -
val_loss: 0.0599 - val_accuracy: 0.9836
Epoch 26/30
60000/60000 [==============================] - 12s 202us/step - loss: 0.0623 - accuracy: 0.9818 -
val_loss: 0.0601 - val_accuracy: 0.9827
Epoch 27/30
60000/60000 [==============================] - 11s 190us/step - loss: 0.0614 - accuracy: 0.9823 -
val_loss: 0.0577 - val_accuracy: 0.9843
Epoch 28/30
60000/60000 [==============================] - 12s 197us/step - loss: 0.0568 - accuracy: 0.9837 -
val_loss: 0.0547 - val_accuracy: 0.9852
Epoch 29/30
60000/60000 [==============================] - 12s 196us/step - loss: 0.0566 - accuracy: 0.9831 -
val_loss: 0.0600 - val_accuracy: 0.9846
Epoch 30/30
60000/60000 [==============================] - 12s 206us/step - loss: 0.0541 - accuracy: 0.9840 -
val_loss: 0.0531 - val_accuracy: 0.9858
```

In [24]:

```
score3 = model3.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score3[0])
print('Test accuracy:', score3[1])

# list of epoch numbers
epoch = list(range(1,nb_epoch+1))
val_loss = history.history['val_loss']
train_loss = history.history['loss']
plt_epoch_vs_loss(epoch, val_loss, train_loss)
```

```
Test score: 0.053111556177656165
Test accuracy: 0.98580002784729
```





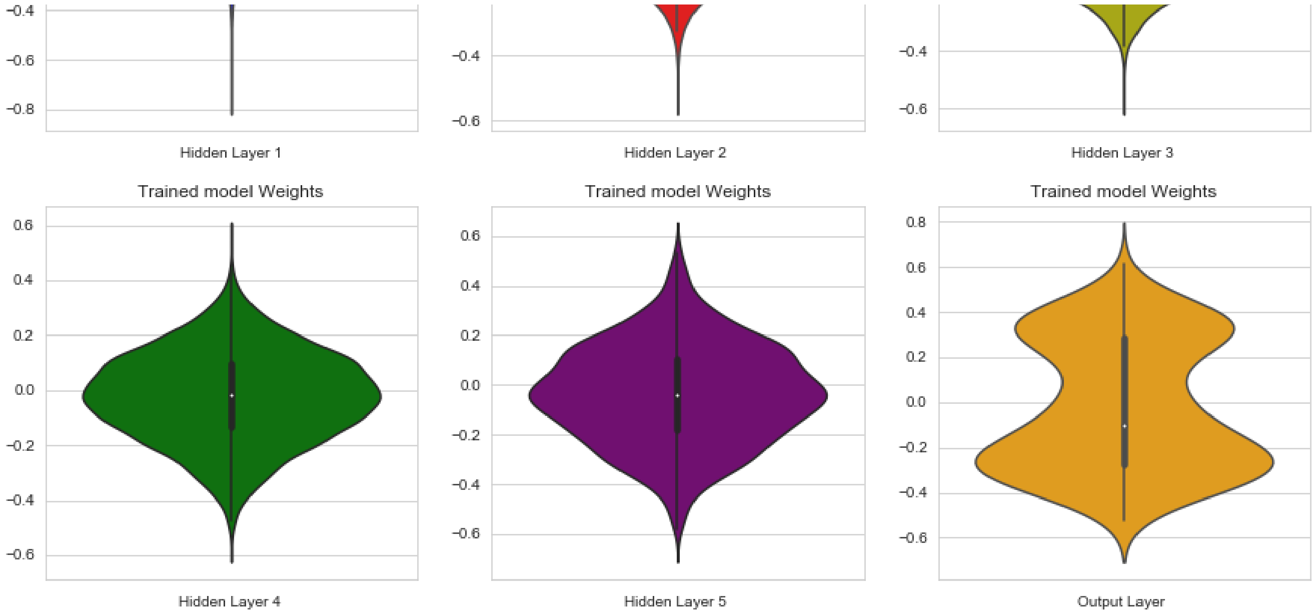
In [25]:

```
w_after = model3.get_weights()
print(len(w_after))
```

32

In [26]:

```
h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[6].flatten().reshape(-1,1)
h3_w = w_after[12].flatten().reshape(-1,1)
h4_w = w_after[18].flatten().reshape(-1,1)
h5_w = w_after[24].flatten().reshape(-1,1)
out_w = w_after[30].flatten().reshape(-1,1)

fig = plt.figure(figsize=(15,10))
plt.title("Weight matrices after model trained")
plt.subplot(2, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

plt.subplot(2, 3, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w,color='y')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(2, 3, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w,color='g')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(2, 3, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h5_w,color='purple')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(2, 3, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='orange')
plt.xlabel('Output Layer ')
plt.show()
```



[3] Results

In [27]:

```
from prettytable import PrettyTable

table = PrettyTable()
table.field_names = ["Model","Hidden Layers","Score","Accuracy"]
table.add_row([1,2,round(score1[0],3),round(score1[1],3)])
table.add_row([2,3,round(score2[0],3),round(score2[1],3)])
table.add_row([3,5,round(score3[0],3),round(score3[1],3)])

print(table.get_string(title="Results"))
```

```
+--------------------------------------------+
|                  Results                   |
+-------+---------------+-------+----------+
| Model | Hidden Layers | Score | Accuracy |
+-------+---------------+-------+----------+
|   1   |       2       | 0.054 |  0.984   |
|   2   |       3       | 0.056 |  0.984   |
|   3   |       5       | 0.053 |  0.986   |
+-------+---------------+-------+----------+
```

[4] Conclusion

There is no much difference in the accuracy of all models.