MLP's on MNIST using Keras

1.0 Loading the MNIST data

In [1]:

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

```
Using TensorFlow backend.
```

In [2]:

```
%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, 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()
```

2.0 Splitting the data

In [3]:

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

```
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```

In [4]:

```
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 [5]:

```
# 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])
```

In [6]:

```
# 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 [7]:

```
# An example data point
print(X_train[0])
```

```
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   3  18  18  18 126 136 175  26 166 255
 247 127   0   0   0   0   0   0   0   0   0   0   0   0  30  36  94 154
 170 253 253 253 253 253 225 172 253 242 195  64   0   0   0   0   0   0
   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93  82
  82  56  39   0   0   0   0   0   0   0   0   0   0   0   0  18 219 253
 253 253 253 253 198 182 247 241   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0  14   1 154 253  90   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0  11 190 253  70   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  35 241
 225 160 108   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0  81 240 253 253 119  25   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0  45 186 253 253 150  27   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253 187
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0 249 253 249  64   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253
 253 207   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0  39 148 229 253 253 253 250 182   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253
 253 201  78   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0  23  66 213 253 253 253 253 198  81   2   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0  18 171 219 253 253 253 253 195
  80   9   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  55 172 226 253 253 253 253 244 133  11   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0 136 253 253 253 212 135 132  16
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0]
```

In [8]:

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

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

In [9]:

```
# example data point after normlizing
print(X_train[0])
```

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

```
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.1372549  0.94509804
 0.88235294 0.62745098 0.42352941 0.00392157 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.31764706 0.94117647 0.99215686
 0.99215686 0.46666667 0.09803922 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.17647059 0.72941176 0.99215686 0.99215686
 0.58823529 0.10588235 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.0627451  0.36470588 0.98823529 0.99215686 0.73333333
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.97647059 0.99215686 0.97647059 0.25098039 0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.18039216 0.50980392 0.71764706 0.99215686
 0.99215686 0.81176471 0.00784314 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.15294118 0.58039216
 0.89803922 0.99215686 0.99215686 0.99215686 0.98039216 0.71372549
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.09411765 0.44705882 0.86666667 0.99215686 0.99215686 0.99215686
 0.99215686 0.78823529 0.30588235 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.09019608 0.25882353 0.83529412 0.99215686
 0.99215686 0.99215686 0.99215686 0.77647059 0.31764706 0.00784314
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.07058824 0.67058824
 0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.76470588
 0.31372549 0.03529412 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.21568627 0.6745098  0.88627451 0.99215686 0.99215686 0.99215686
 0.99215686 0.95686275 0.52156863 0.04313725 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.53333333 0.99215686
 0.99215686 0.99215686 0.83137255 0.52941176 0.51764706 0.0627451
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
```

```
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.        ]
```

In [10]:

```
# 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.]
```

3.0 Softmax classifier

In [0]:

```
# 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 constru
ctor:

# 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_re
gularizer=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
```

In [0]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [0]:

```
# start building a model
model = Sequential()

# The model needs to know what input shape it should expect.
# For this reason, the first layer in a Sequential model
# (and only the first, because following layers can do automatic shape inference)
# needs to receive information about its input shape.
# you can use input_shape and input_dim to pass the shape of input

# output_dim represent the number of nodes need in that layer
# here we have 10 nodes

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

In [0]:

```
# Before training a model, you need to configure the learning process, which is done vi
a the compile method

# It receives three arguments:
# An optimizer. This could be the string identifier of an existing optimizer , https://
keras.io/optimizers/
# A loss function. This is the objective that the model will try to minimize., https://
keras.io/losses/
# A list of metrics. For any classification problem you will want to set this to metric
s=['accuracy'].  https://keras.io/metrics/


# Note: when using the categorical_crossentropy loss, your targets should be in categor
ical format
# (e.g. if you have 10 classes, the target for each sample should be a 10-dimensional v
ector that is all-zeros except
# for a 1 at the index corresponding to the class of the sample).

# that is why we converted out labels into vectors

model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])

# Keras models are trained on Numpy arrays of input data and labels.
# For training a model, you will typically use the  fit function

# fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, valid
ation_split=0.0,
# validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_ep
och=0, steps_per_epoch=None,
# validation_steps=None)

# fit() function Trains the model for a fixed number of epochs (iterations on a datase
t).

# it returns A History object. Its History.history attribute is a record of training lo
ss values and
# metrics values at successive epochs, as well as validation loss values and validation
metrics values (if applicable).

# https://github.com/openai/baselines/issues/20

history = model.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 1/20
60000/60000 [==============================] - 3s 49us/step - loss: 1.2935
- acc: 0.6829 - val_loss: 0.8171 - val_acc: 0.8312
Epoch 2/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.7213
- acc: 0.8385 - val_loss: 0.6099 - val_acc: 0.8623
Epoch 3/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.5904
- acc: 0.8584 - val_loss: 0.5275 - val_acc: 0.8741
Epoch 4/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.5278
- acc: 0.8682 - val_loss: 0.4815 - val_acc: 0.8814
Epoch 5/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.4897
- acc: 0.8747 - val_loss: 0.4515 - val_acc: 0.8870
Epoch 6/20
34432/60000 [================>.............] - ETA: 0s - loss: 0.4697 - ac
c: 0.877360000/60000 [==============================] - 2s 35us/step - los
s: 0.4635 - acc: 0.8799 - val_loss: 0.4303 - val_acc: 0.8898
Epoch 7/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.4442
- acc: 0.8837 - val_loss: 0.4138 - val_acc: 0.8917
Epoch 8/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.4290
- acc: 0.8866 - val_loss: 0.4010 - val_acc: 0.8952
Epoch 9/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.4169
- acc: 0.8894 - val_loss: 0.3909 - val_acc: 0.8971
Epoch 10/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.4067
- acc: 0.8911 - val_loss: 0.3818 - val_acc: 0.8994
Epoch 11/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3982
- acc: 0.8926 - val_loss: 0.3743 - val_acc: 0.9006
Epoch 12/20
 1792/60000 [..............................] - ETA: 1s - loss: 0.4077 - ac
c: 0.889560000/60000 [==============================] - 2s 35us/step - los
s: 0.3908 - acc: 0.8948 - val_loss: 0.3681 - val_acc: 0.9020
Epoch 13/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3844
- acc: 0.8960 - val_loss: 0.3624 - val_acc: 0.9039
Epoch 14/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3786
- acc: 0.8972 - val_loss: 0.3575 - val_acc: 0.9046
Epoch 15/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3735
- acc: 0.8981 - val_loss: 0.3528 - val_acc: 0.9058
Epoch 16/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3689
- acc: 0.8993 - val_loss: 0.3490 - val_acc: 0.9062
Epoch 17/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3648
- acc: 0.9004 - val_loss: 0.3455 - val_acc: 0.9066
Epoch 18/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3610
- acc: 0.9016 - val_loss: 0.3419 - val_acc: 0.9077
Epoch 19/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3575
- acc: 0.9024 - val_loss: 0.3389 - val_acc: 0.9088
Epoch 20/20
```

```
60000/60000 [==============================] - 2s 35us/step - loss: 0.3544
- acc: 0.9032 - val_loss: 0.3362 - val_acc: 0.9093
```

In [0]:

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

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, ve
rbose=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 ep
ochs

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

```
Test score: 0.3362289469957352
Test accuracy: 0.9093
```

4.0 MLP on 784-512-128-10 architecture

4.1 MLP + Sigmoid activation + SGD Optimizer

In [0]:

```
# Multilayer perceptron

model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shape=(input_dim,)))
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_2 (Dense)              (None, 512)               401920    
_________________________________________________________________
dense_3 (Dense)              (None, 128)               65664     
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

In [0]:

```
model_sigmoid.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accur
acy'])

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, v
erbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s 42us/step - loss: 2.2708
- acc: 0.2169 - val_loss: 2.2197 - val_acc: 0.2768
Epoch 2/20
60000/60000 [==============================] - 2s 42us/step - loss: 2.1739
- acc: 0.4237 - val_loss: 2.1143 - val_acc: 0.4877
Epoch 3/20
60000/60000 [==============================] - 2s 42us/step - loss: 2.0506
- acc: 0.5485 - val_loss: 1.9659 - val_acc: 0.5524
Epoch 4/20
60000/60000 [==============================] - 3s 42us/step - loss: 1.8796
- acc: 0.6135 - val_loss: 1.7673 - val_acc: 0.6481
Epoch 5/20
60000/60000 [==============================] - 3s 42us/step - loss: 1.6674
- acc: 0.6659 - val_loss: 1.5414 - val_acc: 0.7205
Epoch 6/20
 5376/60000 [=>............................] - ETA: 2s - loss: 1.5430 - ac
c: 0.698160000/60000 [==============================] - 2s 41us/step - los
s: 1.4449 - acc: 0.7125 - val_loss: 1.3233 - val_acc: 0.7513
Epoch 7/20
60000/60000 [==============================] - 2s 41us/step - loss: 1.2442
- acc: 0.7466 - val_loss: 1.1406 - val_acc: 0.7599
Epoch 8/20
60000/60000 [==============================] - 2s 41us/step - loss: 1.0815
- acc: 0.7722 - val_loss: 0.9974 - val_acc: 0.7968
Epoch 9/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.9544
- acc: 0.7940 - val_loss: 0.8867 - val_acc: 0.8060
Epoch 10/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.8556
- acc: 0.8082 - val_loss: 0.7984 - val_acc: 0.8227
Epoch 11/20
31232/60000 [==============>...............] - ETA: 1s - loss: 0.7920 - ac
c: 0.820760000/60000 [==============================] - 2s 42us/step - los
s: 0.7776 - acc: 0.8225 - val_loss: 0.7292 - val_acc: 0.8335
Epoch 12/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.7154
- acc: 0.8333 - val_loss: 0.6741 - val_acc: 0.8422
Epoch 13/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.6650
- acc: 0.8412 - val_loss: 0.6282 - val_acc: 0.8500
Epoch 14/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.6237
- acc: 0.8485 - val_loss: 0.5906 - val_acc: 0.8585
Epoch 15/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.5893
- acc: 0.8546 - val_loss: 0.5591 - val_acc: 0.8616
Epoch 16/20
34432/60000 [================>.............] - ETA: 0s - loss: 0.5691 - ac
c: 0.857860000/60000 [==============================] - 2s 41us/step - los
s: 0.5606 - acc: 0.8599 - val_loss: 0.5329 - val_acc: 0.8672
Epoch 17/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.5361
- acc: 0.8641 - val_loss: 0.5095 - val_acc: 0.8703
Epoch 18/20
60000/60000 [==============================] - 2s 42us/step - loss: 0.5151
- acc: 0.8676 - val_loss: 0.4904 - val_acc: 0.8736
Epoch 19/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.4969
- acc: 0.8710 - val_loss: 0.4732 - val_acc: 0.8782
```

```
Epoch 20/20
60000/60000 [==============================] - 2s 42us/step - loss: 0.4810
- acc: 0.8739 - val_loss: 0.4583 - val_acc: 0.8816
```

In [0]:

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

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))
vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.4582893396139145
Test accuracy: 0.8816
```

4.2 MLP + Sigmoid activation + ADAM

In [0]:

```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shape=(input_dim,)))
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()

model_sigmoid.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accu
racy'])

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, v
erbose=1, validation_data=(X_test, Y_test))
```

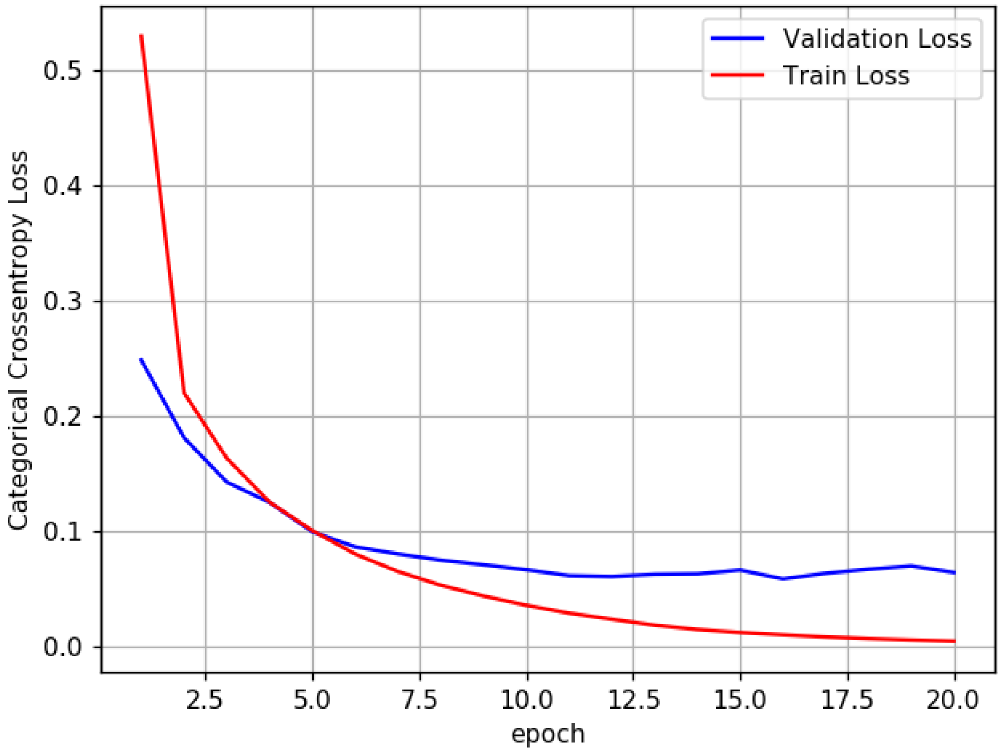
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_5 (Dense)              (None, 512)               401920    
_________________________________________________________________
dense_6 (Dense)              (None, 128)               65664     
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.5308
- acc: 0.8636 - val_loss: 0.2550 - val_acc: 0.9259
Epoch 2/20
53120/60000 [=========================>....] - ETA: 0s - loss: 0.2244 - ac
c: 0.934060000/60000 [==============================] - 3s 51us/step - los
s: 0.2205 - acc: 0.9351 - val_loss: 0.1946 - val_acc: 0.9417
Epoch 3/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.1650
- acc: 0.9512 - val_loss: 0.1421 - val_acc: 0.9570
Epoch 4/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.1279
- acc: 0.9614 - val_loss: 0.1238 - val_acc: 0.9645
Epoch 5/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.1010
- acc: 0.9704 - val_loss: 0.1029 - val_acc: 0.9693
Epoch 6/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0801
- acc: 0.9763 - val_loss: 0.0877 - val_acc: 0.9725
Epoch 7/20
 4480/60000 [=>............................] - ETA: 2s - loss: 0.0632 - ac
c: 0.979560000/60000 [==============================] - 3s 51us/step - los
s: 0.0645 - acc: 0.9809 - val_loss: 0.0831 - val_acc: 0.9751
Epoch 8/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0519
- acc: 0.9842 - val_loss: 0.0724 - val_acc: 0.9780
Epoch 9/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0433
- acc: 0.9872 - val_loss: 0.0714 - val_acc: 0.9786
Epoch 10/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0347
- acc: 0.9898 - val_loss: 0.0695 - val_acc: 0.9776
Epoch 11/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0268
- acc: 0.9930 - val_loss: 0.0659 - val_acc: 0.9796
Epoch 12/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.0219
- acc: 0.9944 - val_loss: 0.0642 - val_acc: 0.9809
Epoch 13/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0180
- acc: 0.9953 - val_loss: 0.0677 - val_acc: 0.9794
Epoch 14/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0133
- acc: 0.9970 - val_loss: 0.0647 - val_acc: 0.9803
Epoch 15/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0114
- acc: 0.9975 - val_loss: 0.0628 - val_acc: 0.9812
```

```
Epoch 16/20
57344/60000 [===========================>..] - ETA: 0s - loss: 0.0085 - ac
c: 0.998260000/60000 [==============================] - 3s 50us/step - los
s: 0.0085 - acc: 0.9982 - val_loss: 0.0666 - val_acc: 0.9806
Epoch 17/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0070
- acc: 0.9986 - val_loss: 0.0643 - val_acc: 0.9822
Epoch 18/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0061
- acc: 0.9986 - val_loss: 0.0656 - val_acc: 0.9818
Epoch 19/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0055
- acc: 0.9988 - val_loss: 0.0811 - val_acc: 0.9774
Epoch 20/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0038
- acc: 0.9992 - val_loss: 0.0723 - val_acc: 0.9818
```

In [0]:

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

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

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

```
Test score: 0.06385514608082886
Test accuracy: 0.9824
```



4.3 MLP + ReLU +SGD

In [0]:

```
# Multilayer perceptron

# https://arxiv.org/pdf/1707.09725.pdf#page=95
# for relu layers
# If we sample weights from a normal distribution N(0,σ) we satisfy this condition with σ=√(2/(ni).
# h1 =>  σ=√(2/(fan_in) = 0.062  => N(0,σ) = N(0,0.062)
# h2 =>  σ=√(2/(fan_in) = 0.125  => N(0,σ) = N(0,0.125)
# out =>  σ=√(2/(fan_in+1) = 0.120  => N(0,σ) = N(0,0.120)

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_8 (Dense)              (None, 512)               401920    
_________________________________________________________________
dense_9 (Dense)              (None, 128)               65664     
_________________________________________________________________
dense_10 (Dense)             (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

In [0]:

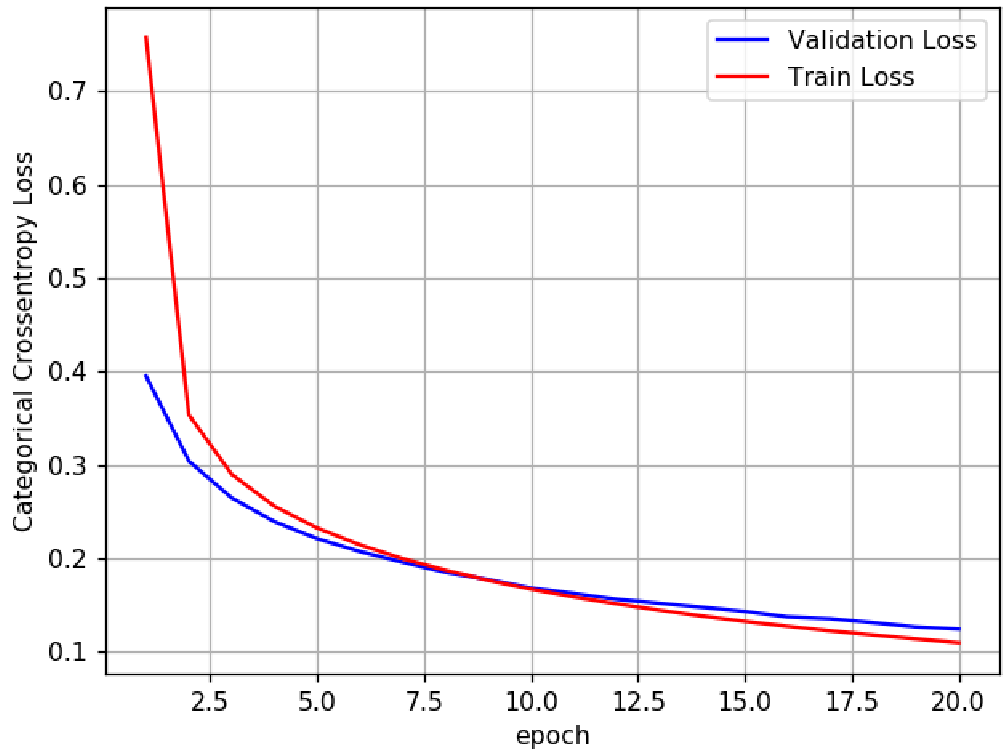
```
model_relu.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.7579
- acc: 0.7812 - val_loss: 0.3951 - val_acc: 0.8921
Epoch 2/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.3535
- acc: 0.8998 - val_loss: 0.3040 - val_acc: 0.9153
Epoch 3/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.2900
- acc: 0.9172 - val_loss: 0.2648 - val_acc: 0.9253
Epoch 4/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.2558
- acc: 0.9269 - val_loss: 0.2393 - val_acc: 0.9316
Epoch 5/20
60000/60000 [==============================] - 4s 58us/step - loss: 0.2324
- acc: 0.9340 - val_loss: 0.2210 - val_acc: 0.9371
Epoch 6/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.2144
- acc: 0.9391 - val_loss: 0.2072 - val_acc: 0.9400
Epoch 7/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1995
- acc: 0.9443 - val_loss: 0.1957 - val_acc: 0.9444
Epoch 8/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1872
- acc: 0.9476 - val_loss: 0.1848 - val_acc: 0.9456
Epoch 9/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.1763
- acc: 0.9507 - val_loss: 0.1771 - val_acc: 0.9488
Epoch 10/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.1668
- acc: 0.9539 - val_loss: 0.1682 - val_acc: 0.9506
Epoch 11/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.1585
- acc: 0.9560 - val_loss: 0.1623 - val_acc: 0.9518
Epoch 12/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.151
1 - acc: 0.9577 - val_loss: 0.1560 - val_acc: 0.9543
Epoch 13/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.144
3 - acc: 0.9596 - val_loss: 0.1517 - val_acc: 0.9557
Epoch 14/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.137
9 - acc: 0.9615 - val_loss: 0.1474 - val_acc: 0.9572
Epoch 15/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1323
- acc: 0.9628 - val_loss: 0.1429 - val_acc: 0.9580
Epoch 16/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1270
- acc: 0.9645 - val_loss: 0.1371 - val_acc: 0.9598
Epoch 17/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.122
1 - acc: 0.9661 - val_loss: 0.1351 - val_acc: 0.9602
Epoch 18/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1177
- acc: 0.9671 - val_loss: 0.1309 - val_acc: 0.9618
Epoch 19/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.1136
- acc: 0.9685 - val_loss: 0.1263 - val_acc: 0.9631
Epoch 20/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.1094
- acc: 0.9694 - val_loss: 0.1241 - val_acc: 0.9631
```

In [0]:

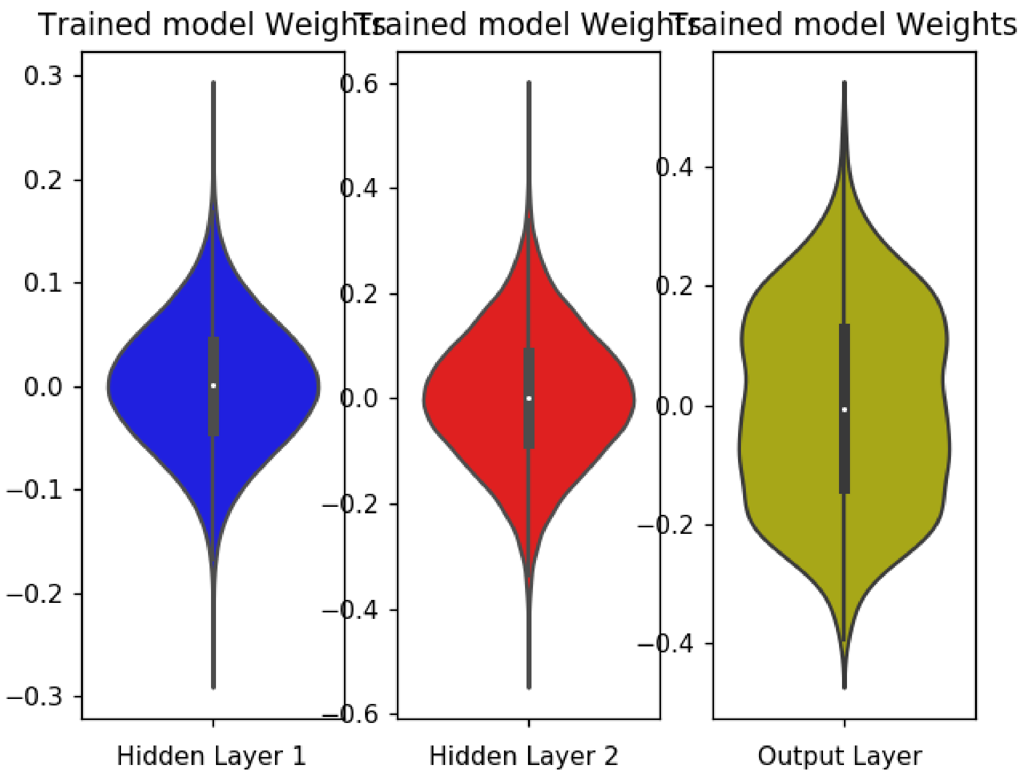
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

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

```
Test score: 0.12405014228336513
Test accuracy: 0.9631
```



In [0]:

```
w_after = model_relu.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



4.4 MLP + ReLU + ADAM

In [0]:

```
model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0,
stddev=0.125, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

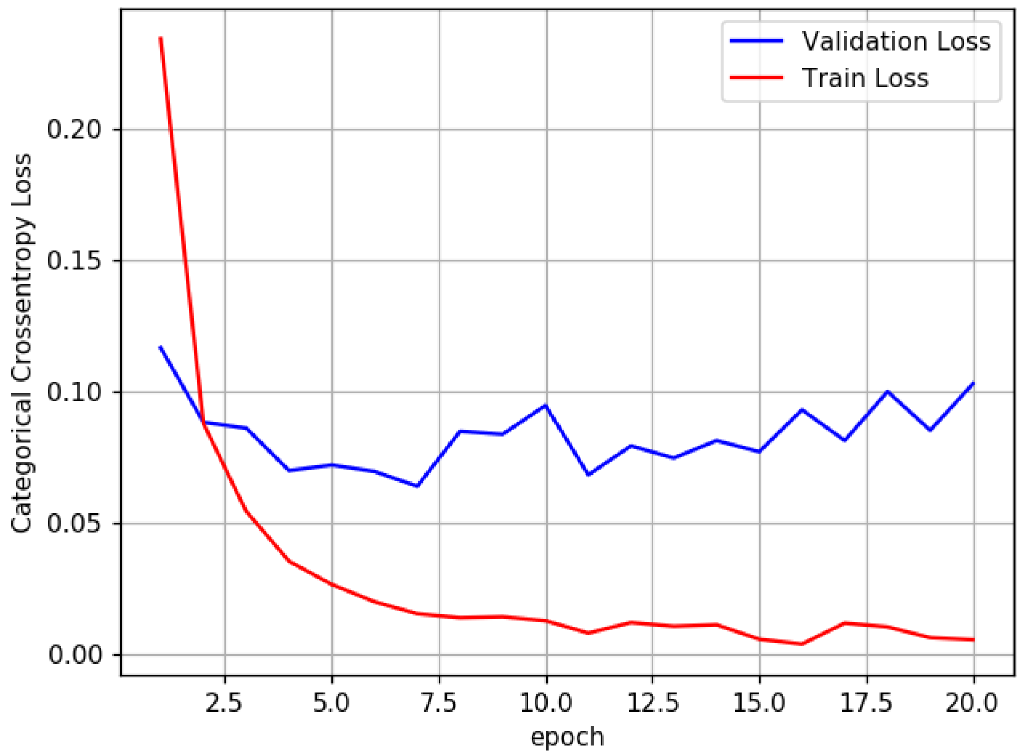
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_11 (Dense)             (None, 512)               401920    
_________________________________________________________________
dense_12 (Dense)             (None, 128)               65664     
_________________________________________________________________
dense_13 (Dense)             (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.234
1 - acc: 0.9295 - val_loss: 0.1165 - val_acc: 0.9652
Epoch 2/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0878
- acc: 0.9729 - val_loss: 0.0883 - val_acc: 0.9720
Epoch 3/20
60000/60000 [==============================] - 5s 75us/step - loss: 0.0544
- acc: 0.9825 - val_loss: 0.0860 - val_acc: 0.9729
Epoch 4/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0354
- acc: 0.9885 - val_loss: 0.0699 - val_acc: 0.9797
Epoch 5/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0266
- acc: 0.9914 - val_loss: 0.0720 - val_acc: 0.9788
Epoch 6/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0200
- acc: 0.9941 - val_loss: 0.0696 - val_acc: 0.9803
Epoch 7/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0155
- acc: 0.9951 - val_loss: 0.0640 - val_acc: 0.9829
Epoch 8/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0140
- acc: 0.9952 - val_loss: 0.0848 - val_acc: 0.9792
Epoch 9/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0143
- acc: 0.9952 - val_loss: 0.0837 - val_acc: 0.9796
Epoch 10/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.012
8 - acc: 0.9958 - val_loss: 0.0946 - val_acc: 0.9782
Epoch 11/20
60000/60000 [==============================] - 7s 125us/step - loss: 0.008
1 - acc: 0.9974 - val_loss: 0.0682 - val_acc: 0.9826
Epoch 12/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.012
1 - acc: 0.9959 - val_loss: 0.0793 - val_acc: 0.9816
Epoch 13/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.010
7 - acc: 0.9963 - val_loss: 0.0746 - val_acc: 0.9820
Epoch 14/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.011
3 - acc: 0.9960 - val_loss: 0.0813 - val_acc: 0.9816
Epoch 15/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0058
- acc: 0.9982 - val_loss: 0.0770 - val_acc: 0.9842
Epoch 16/20
```

```
60000/60000 [==============================] - 4s 65us/step - loss: 0.0040
- acc: 0.9987 - val_loss: 0.0930 - val_acc: 0.9808
Epoch 17/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0119
- acc: 0.9959 - val_loss: 0.0813 - val_acc: 0.9819
Epoch 18/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0105
- acc: 0.9966 - val_loss: 0.1000 - val_acc: 0.9803
Epoch 19/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0064
- acc: 0.9981 - val_loss: 0.0852 - val_acc: 0.9831
Epoch 20/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0056
- acc: 0.9982 - val_loss: 0.1029 - val_acc: 0.9805
```

In [0]:

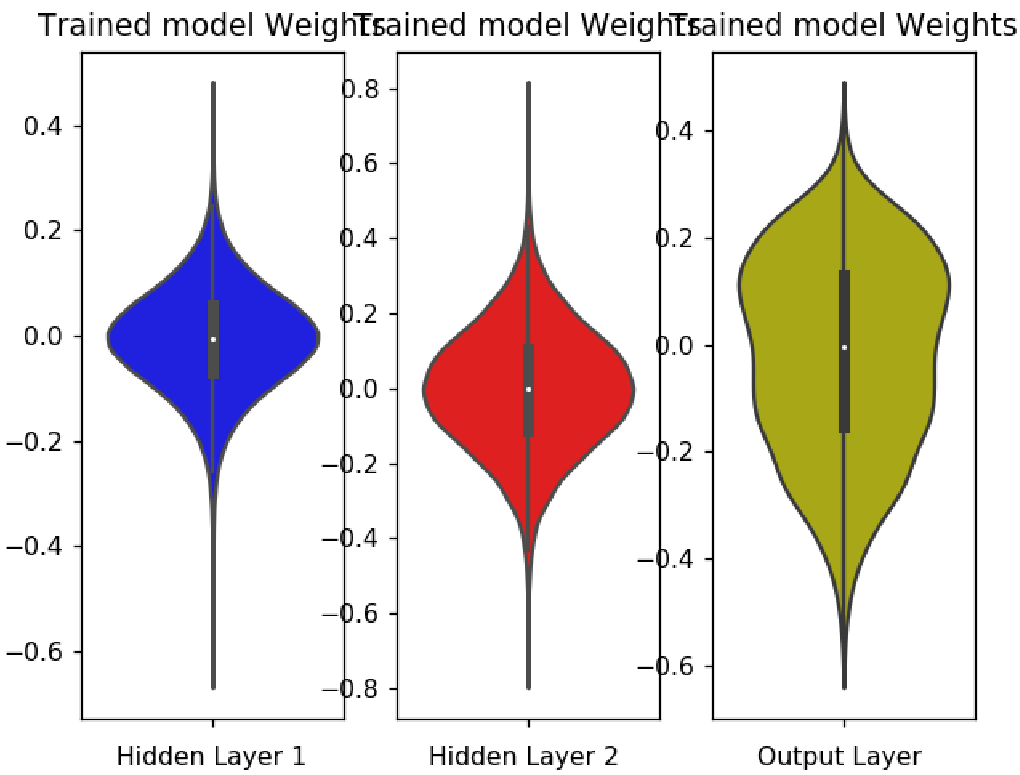
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

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

```
Test score: 0.10294274219236926
Test accuracy: 0.9805
```



In [0]:

```
w_after = model_relu.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



4.5 MLP + Batch-Norm on hidden Layers + AdamOptimizer </2>

In [0]:

```
# 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_batch = Sequential()

model_batch.add(Dense(512, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(128, activation='sigmoid', kernel_initializer=RandomNormal(mean=
0.0, stddev=0.55, seed=None)) )
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_14 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048      
_________________________________________________________________
dense_15 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512       
_________________________________________________________________
dense_16 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [0]:

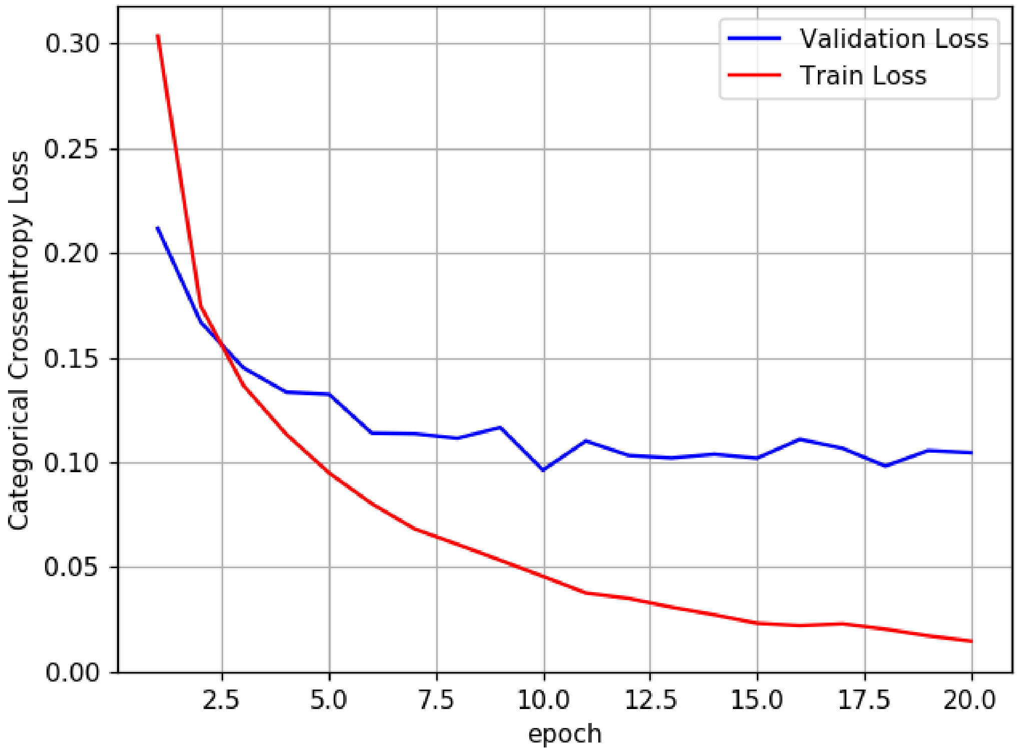
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura
cy'])

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, ver
bose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.303
6 - acc: 0.9104 - val_loss: 0.2116 - val_acc: 0.9376
Epoch 2/20
60000/60000 [==============================] - 10s 170us/step - loss: 0.17
47 - acc: 0.9483 - val_loss: 0.1670 - val_acc: 0.9505
Epoch 3/20
60000/60000 [==============================] - 13s 220us/step - loss: 0.13
67 - acc: 0.9599 - val_loss: 0.1451 - val_acc: 0.9567
Epoch 4/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.113
4 - acc: 0.9666 - val_loss: 0.1335 - val_acc: 0.9603
Epoch 5/20
60000/60000 [==============================] - 13s 211us/step - loss: 0.09
49 - acc: 0.9703 - val_loss: 0.1325 - val_acc: 0.9589
Epoch 6/20
60000/60000 [==============================] - 7s 119us/step - loss: 0.080
2 - acc: 0.9758 - val_loss: 0.1139 - val_acc: 0.9652
Epoch 7/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.068
2 - acc: 0.9787 - val_loss: 0.1136 - val_acc: 0.9666
Epoch 8/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.060
8 - acc: 0.9815 - val_loss: 0.1114 - val_acc: 0.9666
Epoch 9/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.053
2 - acc: 0.9837 - val_loss: 0.1167 - val_acc: 0.9666
Epoch 10/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.045
5 - acc: 0.9856 - val_loss: 0.0962 - val_acc: 0.9718
Epoch 11/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.037
6 - acc: 0.9880 - val_loss: 0.1102 - val_acc: 0.9673
Epoch 12/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.035
0 - acc: 0.9889 - val_loss: 0.1033 - val_acc: 0.9710
Epoch 13/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.030
8 - acc: 0.9903 - val_loss: 0.1020 - val_acc: 0.9712
Epoch 14/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.027
1 - acc: 0.9913 - val_loss: 0.1038 - val_acc: 0.9727
Epoch 15/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.023
1 - acc: 0.9926 - val_loss: 0.1019 - val_acc: 0.9717
Epoch 16/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.022
0 - acc: 0.9928 - val_loss: 0.1110 - val_acc: 0.9703
Epoch 17/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.022
9 - acc: 0.9928 - val_loss: 0.1067 - val_acc: 0.9739
Epoch 18/20
60000/60000 [==============================] - 8s 128us/step - loss: 0.020
3 - acc: 0.9935 - val_loss: 0.0982 - val_acc: 0.9738
Epoch 19/20
60000/60000 [==============================] - 7s 125us/step - loss: 0.017
1 - acc: 0.9944 - val_loss: 0.1056 - val_acc: 0.9706
Epoch 20/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.01
46 - acc: 0.9952 - val_loss: 0.1046 - val_acc: 0.9732
```

In [0]:

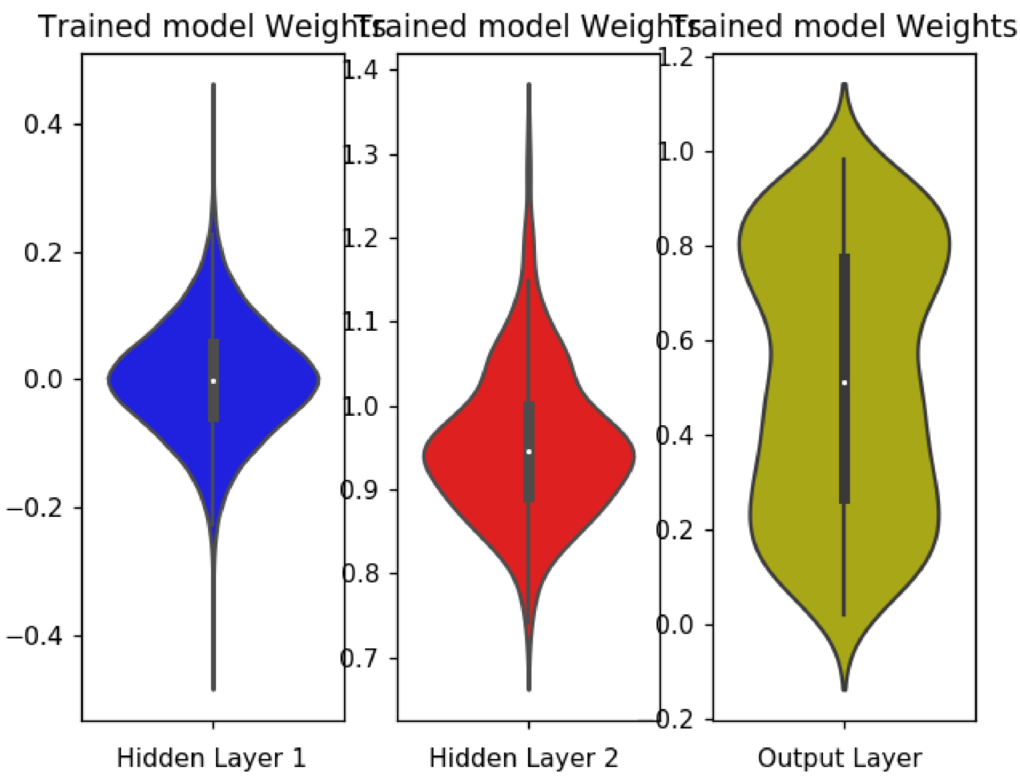
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

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

```
Test score: 0.10456635547156475
Test accuracy: 0.9732
```



In [0]:

```
w_after = model_batch.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



4.6 MLP + Dropout + AdamOptimizer

In [0]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='sigmoid', input_shape=(input_dim,), kernel_initia
lizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.
0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_17 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_18 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_4 (Batch (None, 128)               512       
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_19 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [0]:

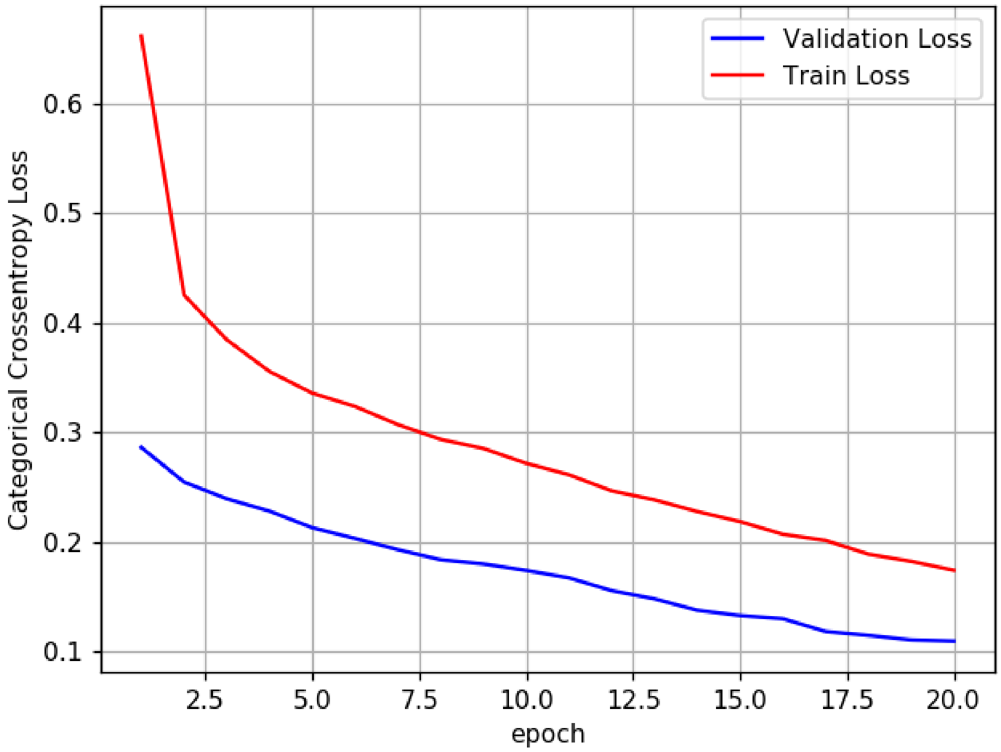
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 14s 227us/step - loss: 0.66
12 - acc: 0.7951 - val_loss: 0.2860 - val_acc: 0.9166
Epoch 2/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.425
0 - acc: 0.8710 - val_loss: 0.2545 - val_acc: 0.9252
Epoch 3/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.38
41 - acc: 0.8846 - val_loss: 0.2391 - val_acc: 0.9298
Epoch 4/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.355
1 - acc: 0.8927 - val_loss: 0.2279 - val_acc: 0.9325
Epoch 5/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.335
5 - acc: 0.8986 - val_loss: 0.2127 - val_acc: 0.9356
Epoch 6/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.323
4 - acc: 0.9031 - val_loss: 0.2029 - val_acc: 0.9387: 1s - loss:
Epoch 7/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.306
8 - acc: 0.9077 - val_loss: 0.1927 - val_acc: 0.9421
Epoch 8/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.29
33 - acc: 0.9113 - val_loss: 0.1836 - val_acc: 0.9453
Epoch 9/20
60000/60000 [==============================] - 13s 222us/step - loss: 0.28
50 - acc: 0.9131 - val_loss: 0.1797 - val_acc: 0.9451
Epoch 10/20
60000/60000 [==============================] - 14s 236us/step - loss: 0.27
15 - acc: 0.9187 - val_loss: 0.1738 - val_acc: 0.9465
Epoch 11/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.261
1 - acc: 0.9214 - val_loss: 0.1671 - val_acc: 0.9506
Epoch 12/20
60000/60000 [==============================] - 8s 134us/step - loss: 0.246
4 - acc: 0.9252 - val_loss: 0.1554 - val_acc: 0.9525
Epoch 13/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.238
2 - acc: 0.9278 - val_loss: 0.1479 - val_acc: 0.9554
Epoch 14/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.227
5 - acc: 0.9313 - val_loss: 0.1375 - val_acc: 0.9580
Epoch 15/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.218
3 - acc: 0.9337 - val_loss: 0.1326 - val_acc: 0.9599
Epoch 16/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.206
8 - acc: 0.9384 - val_loss: 0.1297 - val_acc: 0.9613 loss: 0.2066 - ac
Epoch 17/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.201
1 - acc: 0.9395 - val_loss: 0.1181 - val_acc: 0.9646
Epoch 18/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.188
6 - acc: 0.9435 - val_loss: 0.1145 - val_acc: 0.9658
Epoch 19/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.182
1 - acc: 0.9451 - val_loss: 0.1104 - val_acc: 0.9662
Epoch 20/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.173
9 - acc: 0.9473 - val_loss: 0.1093 - val_acc: 0.9679
```

In [0]:

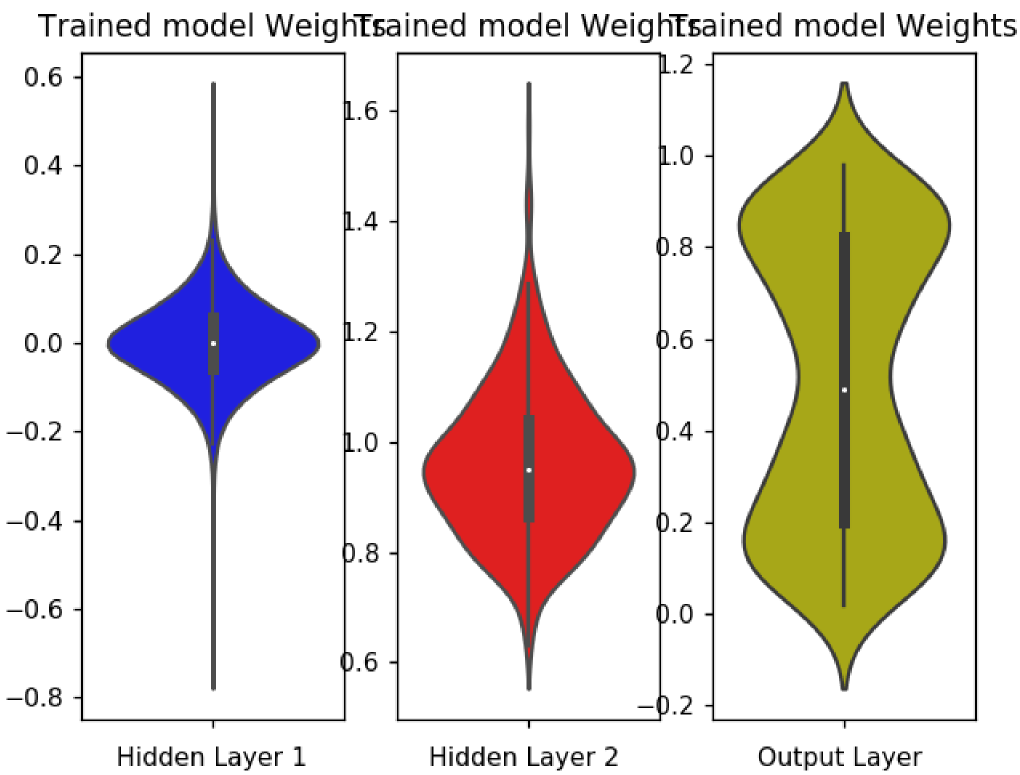
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

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

```
Test score: 0.1093290721397847
Test accuracy: 0.9679
```



In [0]:

```
w_after = model_drop.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



4.7 Hyper-parameter tuning of Keras models using Sklearn

In [0]:

```
from keras.optimizers import Adam,RMSprop,SGD
def best_hyperparameters(activ):

    model = Sequential()
    model.add(Dense(512, activation=activ, input_shape=(input_dim,), kernel_initializer
=RandomNormal(mean=0.0, stddev=0.062, seed=None)))
    model.add(Dense(128, activation=activ, kernel_initializer=RandomNormal(mean=0.0, st
ddev=0.125, seed=None)) )
    model.add(Dense(output_dim, activation='softmax'))


    model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='ada
m')

    return model
```

In [0]:

```
# https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-p
ython-keras/

activ = ['sigmoid','relu']

from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV

model = KerasClassifier(build_fn=best_hyperparameters, epochs=nb_epoch, batch_size=batc
h_size, verbose=0)
param_grid = dict(activ=activ)

# if you are using CPU
# grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
# if you are using GPU dont use the n_jobs parameter

grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid_result = grid.fit(X_train, Y_train)
```

In [0]:

```
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.975633 using {'activ': 'relu'}
0.974650 (0.001138) with: {'activ': 'sigmoid'}
0.975633 (0.002812) with: {'activ': 'relu'}
```

5.0 Assignment

1. Please try out models with different architectures & you can experiment with number of layers as well.
2. Include error plots.
3. Compare all your models in a tabular format using prettytable library or similar ones.

5.1 Defining model parameters

In [12]:

```
output_dim = 10
input_dim = X_train.shape[1]

batch_size = 200
nb_epoch = 30
```

In [13]:

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
import time
from keras.models import Sequential
from keras.layers import Dense, Activation
from datetime import datetime
# 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, 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()
```

5.2 Two hidden layers with 784-400-80-10 architecture

5.2.1 Without Drop-out & Batch Normalization

In [16]:

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

start = datetime.now()

model1 = Sequential()
model1.add(Dense(400, activation='relu', input_shape=(input_dim,), kernel_initializer=
'he_normal'))
model1.add(Dense(80, activation='relu', kernel_initializer= 'he_normal'))
model1.add(Dense(output_dim, activation='softmax'))

print(model1.summary())

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, validation_data=(X_test, Y_test))

print('Time taken :', datetime.now() - start)
```

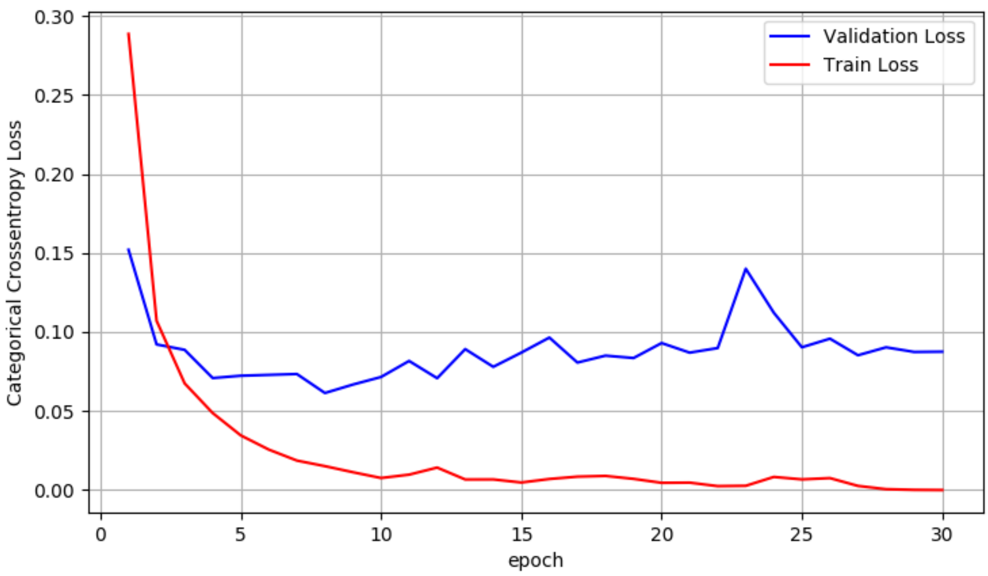
```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_7 (Dense)              (None, 400)               314000    
_________________________________________________________________
dense_8 (Dense)              (None, 80)                32080     
_________________________________________________________________
dense_9 (Dense)              (None, 10)                810       
=================================================================
Total params: 346,890
Trainable params: 346,890
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 7s 110us/step - loss: 0.288
5 - accuracy: 0.9172 - val_loss: 0.1520 - val_accuracy: 0.9546
Epoch 2/30
60000/60000 [==============================] - 6s 94us/step - loss: 0.1071
- accuracy: 0.9677 - val_loss: 0.0921 - val_accuracy: 0.9706
Epoch 3/30
60000/60000 [==============================] - 6s 99us/step - loss: 0.0675
- accuracy: 0.9795 - val_loss: 0.0887 - val_accuracy: 0.9718
Epoch 4/30
60000/60000 [==============================] - 6s 98us/step - loss: 0.0487
- accuracy: 0.9850 - val_loss: 0.0709 - val_accuracy: 0.9784
Epoch 5/30
60000/60000 [==============================] - 6s 97us/step - loss: 0.0346
- accuracy: 0.9891 - val_loss: 0.0723 - val_accuracy: 0.9783
Epoch 6/30
60000/60000 [==============================] - 6s 100us/step - loss: 0.025
6 - accuracy: 0.9919 - val_loss: 0.0729 - val_accuracy: 0.9776
Epoch 7/30
60000/60000 [==============================] - 6s 100us/step - loss: 0.018
7 - accuracy: 0.9947 - val_loss: 0.0734 - val_accuracy: 0.9785
Epoch 8/30
60000/60000 [==============================] - 6s 106us/step - loss: 0.015
2 - accuracy: 0.9955 - val_loss: 0.0614 - val_accuracy: 0.9808
Epoch 9/30
60000/60000 [==============================] - 6s 101us/step - loss: 0.011
3 - accuracy: 0.9969 - val_loss: 0.0668 - val_accuracy: 0.9818
Epoch 10/30
60000/60000 [==============================] - 6s 105us/step - loss: 0.007
7 - accuracy: 0.9981 - val_loss: 0.0715 - val_accuracy: 0.9797
Epoch 11/30
60000/60000 [==============================] - 6s 96us/step - loss: 0.0098
- accuracy: 0.9968 - val_loss: 0.0817 - val_accuracy: 0.9781
Epoch 12/30
60000/60000 [==============================] - 6s 103us/step - loss: 0.014
3 - accuracy: 0.9948 - val_loss: 0.0708 - val_accuracy: 0.9803
Epoch 13/30
60000/60000 [==============================] - 6s 104us/step - loss: 0.006
7 - accuracy: 0.9982 - val_loss: 0.0892 - val_accuracy: 0.9784
Epoch 14/30
60000/60000 [==============================] - 6s 96us/step - loss: 0.0068
- accuracy: 0.9979 - val_loss: 0.0780 - val_accuracy: 0.9811
Epoch 15/30
60000/60000 [==============================] - 6s 108us/step - loss: 0.004
8 - accuracy: 0.9984 - val_loss: 0.0870 - val_accuracy: 0.9804
```

```
Epoch 16/30
60000/60000 [==============================] - 6s 95us/step - loss: 0.0070
- accuracy: 0.9977 - val_loss: 0.0965 - val_accuracy: 0.9780
Epoch 17/30
60000/60000 [==============================] - 6s 102us/step - loss: 0.008
5 - accuracy: 0.9969 - val_loss: 0.0807 - val_accuracy: 0.9808
Epoch 18/30
60000/60000 [==============================] - 6s 98us/step - loss: 0.0090
- accuracy: 0.9969 - val_loss: 0.0851 - val_accuracy: 0.9814
Epoch 19/30
60000/60000 [==============================] - 6s 96us/step - loss: 0.0071
- accuracy: 0.9978 - val_loss: 0.0835 - val_accuracy: 0.9808
Epoch 20/30
60000/60000 [==============================] - 6s 94us/step - loss: 0.0046
- accuracy: 0.9987 - val_loss: 0.0930 - val_accuracy: 0.9803
Epoch 21/30
60000/60000 [==============================] - 6s 99us/step - loss: 0.0047
- accuracy: 0.9985 - val_loss: 0.0869 - val_accuracy: 0.9822
Epoch 22/30
60000/60000 [==============================] - 6s 105us/step - loss: 0.002
5 - accuracy: 0.9994 - val_loss: 0.0898 - val_accuracy: 0.9817
Epoch 23/30
60000/60000 [==============================] - 6s 107us/step - loss: 0.002
7 - accuracy: 0.9991 - val_loss: 0.1400 - val_accuracy: 0.9744
Epoch 24/30
60000/60000 [==============================] - 6s 98us/step - loss: 0.0083
- accuracy: 0.9971 - val_loss: 0.1122 - val_accuracy: 0.9784
Epoch 25/30
60000/60000 [==============================] - 6s 102us/step - loss: 0.006
8 - accuracy: 0.9977 - val_loss: 0.0903 - val_accuracy: 0.9824
Epoch 26/30
60000/60000 [==============================] - 6s 99us/step - loss: 0.0076
- accuracy: 0.9975 - val_loss: 0.0958 - val_accuracy: 0.9822
Epoch 27/30
60000/60000 [==============================] - 7s 110us/step - loss: 0.002
6 - accuracy: 0.9992 - val_loss: 0.0853 - val_accuracy: 0.9839
Epoch 28/30
60000/60000 [==============================] - 6s 107us/step - loss: 5.944
6e-04 - accuracy: 0.9999 - val_loss: 0.0903 - val_accuracy: 0.9834
Epoch 29/30
60000/60000 [==============================] - 6s 97us/step - loss: 1.5903
e-04 - accuracy: 1.0000 - val_loss: 0.0874 - val_accuracy: 0.9844
Epoch 30/30
60000/60000 [==============================] - 6s 97us/step - loss: 4.9647
e-05 - accuracy: 1.0000 - val_loss: 0.0875 - val_accuracy: 0.9848
Time taken : 0:03:02.169499
```

In [17]:

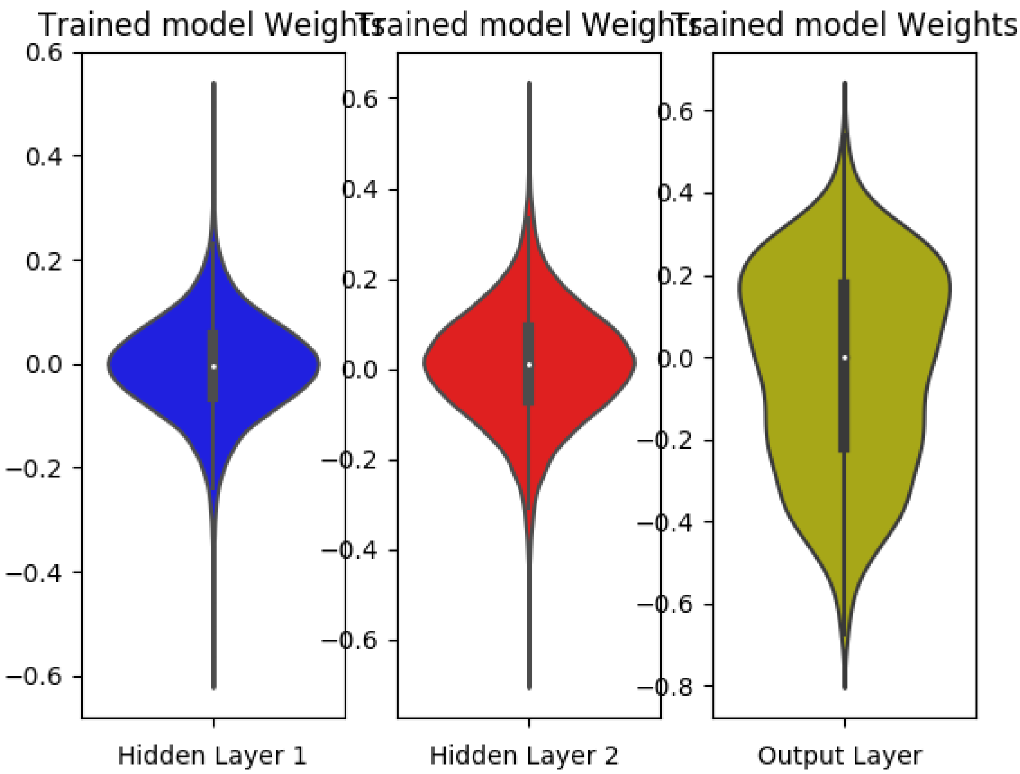
```
score1 = model1.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score1[0])
print('Test accuracy:', score1[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, ve
rbose=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 ep
ochs


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

```
Test score: 0.08754952983312075
Test accuracy: 0.9847999811172485
```



In [18]:

```
w_after = model1.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



5.2.2 With Drop-out & Batch Normalization

In [21]:

```
import warnings
warnings.filterwarnings("ignore")
start = datetime.now()

from keras.layers.normalization import BatchNormalization
from keras.layers import Dropout

model2 = Sequential()

model2.add(Dense(400, activation='relu', input_shape=(input_dim,), kernel_initializer=
'he_normal'))
model2.add(BatchNormalization())
model2.add(Dropout(0.5))

model2.add(Dense(80, activation='relu', kernel_initializer= 'he_normal'))
model2.add(BatchNormalization())
model2.add(Dropout(0.5))

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

print(model2.summary())

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, validation_data=(X_test, Y_test))

print('Time taken :', datetime.now() - start)
```

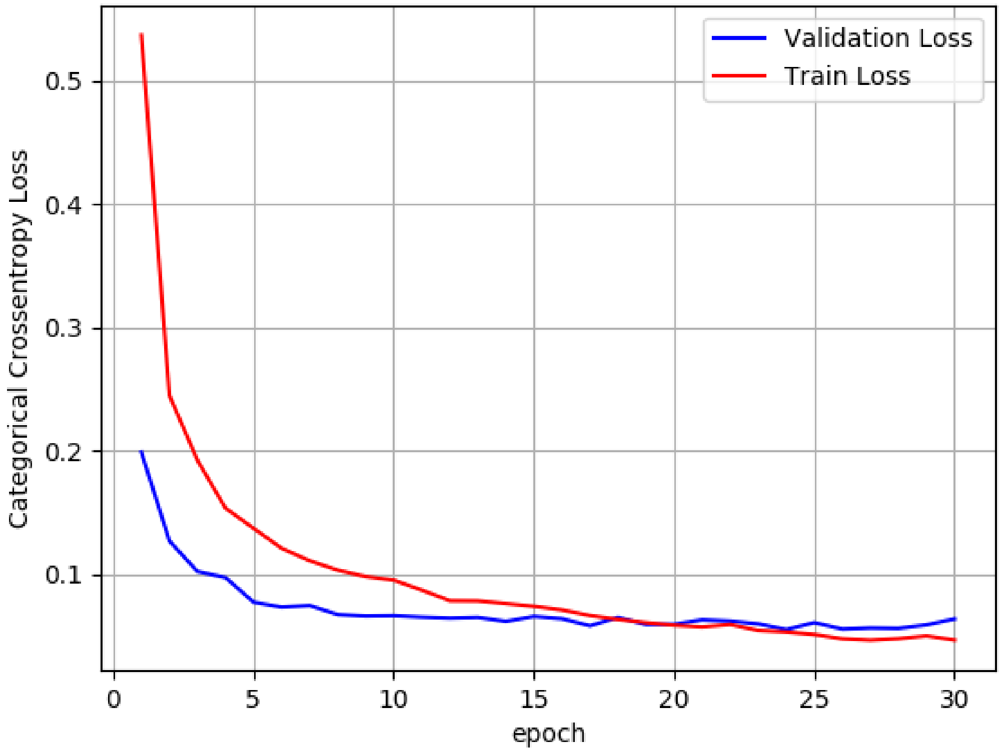
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_12 (Dense)             (None, 400)               314000    
_________________________________________________________________
batch_normalization_2 (Batch (None, 400)               1600      
_________________________________________________________________
dropout_1 (Dropout)          (None, 400)               0         
_________________________________________________________________
dense_13 (Dense)             (None, 80)                32080     
_________________________________________________________________
batch_normalization_3 (Batch (None, 80)                320       
_________________________________________________________________
dropout_2 (Dropout)          (None, 80)                0         
_________________________________________________________________
dense_14 (Dense)             (None, 10)                810       
=================================================================
Total params: 348,810
Trainable params: 347,850
Non-trainable params: 960
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 12s 203us/step - loss: 0.53
73 - accuracy: 0.8386 - val_loss: 0.1991 - val_accuracy: 0.9438
Epoch 2/30
60000/60000 [==============================] - 9s 153us/step - loss: 0.245
0 - accuracy: 0.9280 - val_loss: 0.1270 - val_accuracy: 0.9627
Epoch 3/30
60000/60000 [==============================] - 10s 162us/step - loss: 0.19
25 - accuracy: 0.9435 - val_loss: 0.1023 - val_accuracy: 0.9683
Epoch 4/30
60000/60000 [==============================] - 9s 157us/step - loss: 0.153
7 - accuracy: 0.9549 - val_loss: 0.0974 - val_accuracy: 0.9716
Epoch 5/30
60000/60000 [==============================] - 10s 161us/step - loss: 0.13
72 - accuracy: 0.9600 - val_loss: 0.0774 - val_accuracy: 0.9756
Epoch 6/30
60000/60000 [==============================] - 9s 158us/step - loss: 0.121
1 - accuracy: 0.9636 - val_loss: 0.0734 - val_accuracy: 0.9761
Epoch 7/30
60000/60000 [==============================] - 10s 165us/step - loss: 0.11
10 - accuracy: 0.9676 - val_loss: 0.0746 - val_accuracy: 0.9763
Epoch 8/30
60000/60000 [==============================] - 9s 147us/step - loss: 0.103
4 - accuracy: 0.9682 - val_loss: 0.0673 - val_accuracy: 0.9788
Epoch 9/30
60000/60000 [==============================] - 9s 143us/step - loss: 0.098
2 - accuracy: 0.9704 - val_loss: 0.0661 - val_accuracy: 0.9812
Epoch 10/30
60000/60000 [==============================] - 8s 142us/step - loss: 0.095
2 - accuracy: 0.9704 - val_loss: 0.0664 - val_accuracy: 0.9799
Epoch 11/30
60000/60000 [==============================] - 9s 149us/step - loss: 0.087
2 - accuracy: 0.9732 - val_loss: 0.0652 - val_accuracy: 0.9793
Epoch 12/30
60000/60000 [==============================] - 9s 148us/step - loss: 0.078
6 - accuracy: 0.9759 - val_loss: 0.0644 - val_accuracy: 0.9799
Epoch 13/30
```

```
60000/60000 [==============================] - 9s 143us/step - loss: 0.078
4 - accuracy: 0.9764 - val_loss: 0.0650 - val_accuracy: 0.9798
Epoch 14/30
60000/60000 [==============================] - 9s 142us/step - loss: 0.076
3 - accuracy: 0.9770 - val_loss: 0.0617 - val_accuracy: 0.9822
Epoch 15/30
60000/60000 [==============================] - 9s 150us/step - loss: 0.074
0 - accuracy: 0.9773 - val_loss: 0.0658 - val_accuracy: 0.9817
Epoch 16/30
60000/60000 [==============================] - 9s 145us/step - loss: 0.071
1 - accuracy: 0.9783 - val_loss: 0.0639 - val_accuracy: 0.9807
Epoch 17/30
60000/60000 [==============================] - 9s 152us/step - loss: 0.066
5 - accuracy: 0.9794 - val_loss: 0.0582 - val_accuracy: 0.9828
Epoch 18/30
60000/60000 [==============================] - 9s 144us/step - loss: 0.063
4 - accuracy: 0.9805 - val_loss: 0.0650 - val_accuracy: 0.9807
Epoch 19/30
60000/60000 [==============================] - 9s 143us/step - loss: 0.060
7 - accuracy: 0.9804 - val_loss: 0.0592 - val_accuracy: 0.9829
Epoch 20/30
60000/60000 [==============================] - 8s 137us/step - loss: 0.058
6 - accuracy: 0.9815 - val_loss: 0.0594 - val_accuracy: 0.9825
Epoch 21/30
60000/60000 [==============================] - 8s 135us/step - loss: 0.057
3 - accuracy: 0.9818 - val_loss: 0.0630 - val_accuracy: 0.9826
Epoch 22/30
60000/60000 [==============================] - 9s 142us/step - loss: 0.059
2 - accuracy: 0.9808 - val_loss: 0.0620 - val_accuracy: 0.9820
Epoch 23/30
60000/60000 [==============================] - 9s 143us/step - loss: 0.054
4 - accuracy: 0.9827 - val_loss: 0.0598 - val_accuracy: 0.9839
Epoch 24/30
60000/60000 [==============================] - 8s 141us/step - loss: 0.053
2 - accuracy: 0.9830 - val_loss: 0.0552 - val_accuracy: 0.9847
Epoch 25/30
60000/60000 [==============================] - 9s 145us/step - loss: 0.051
2 - accuracy: 0.9844 - val_loss: 0.0608 - val_accuracy: 0.9832
Epoch 26/30
60000/60000 [==============================] - 8s 140us/step - loss: 0.047
7 - accuracy: 0.9849 - val_loss: 0.0556 - val_accuracy: 0.9851
Epoch 27/30
60000/60000 [==============================] - 8s 136us/step - loss: 0.046
6 - accuracy: 0.9855 - val_loss: 0.0564 - val_accuracy: 0.9850
Epoch 28/30
60000/60000 [==============================] - 9s 143us/step - loss: 0.047
8 - accuracy: 0.9851 - val_loss: 0.0561 - val_accuracy: 0.9843
Epoch 29/30
60000/60000 [==============================] - 8s 140us/step - loss: 0.050
0 - accuracy: 0.9846 - val_loss: 0.0589 - val_accuracy: 0.9834
Epoch 30/30
60000/60000 [==============================] - 8s 132us/step - loss: 0.046
7 - accuracy: 0.9853 - val_loss: 0.0637 - val_accuracy: 0.9825
Time taken : 0:04:28.345052
```

In [22]:

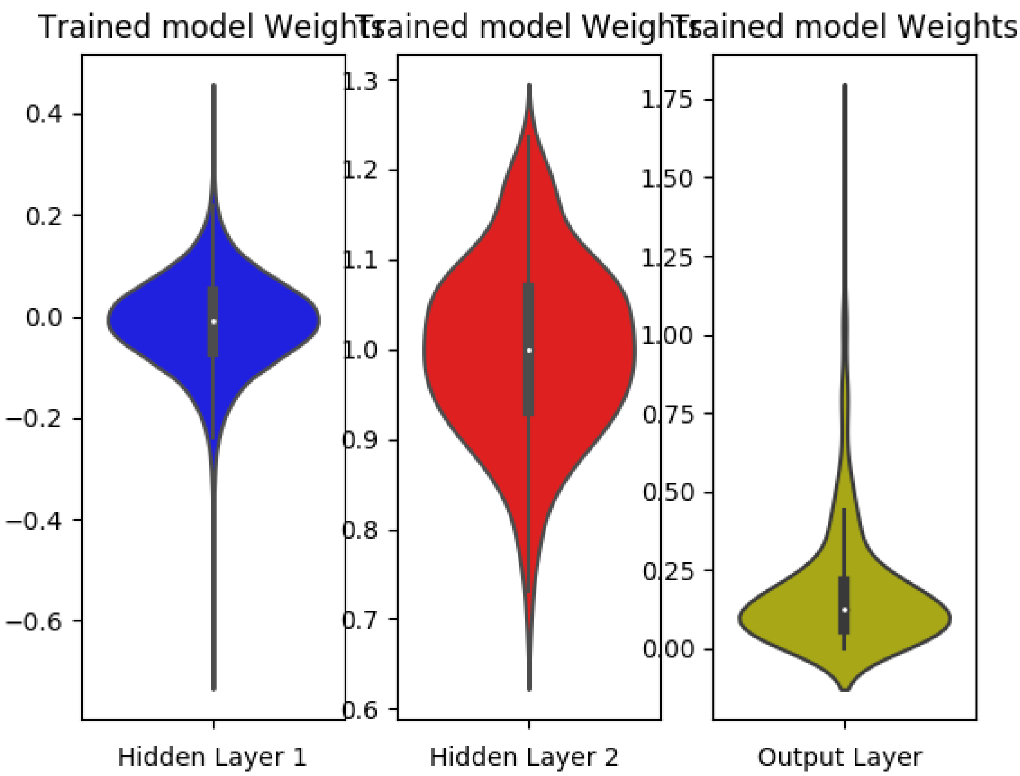
```
score2 = model2.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score2[0])
print('Test accuracy:', score2[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

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

```
Test score: 0.06374456858527555
Test accuracy: 0.9825000166893005
```



In [23]:

```
w_after = model2.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



5.3 Three hidden layers with 784-450-200-90-10 architecture

5.3.1 Without Drop-out & Batch Normalization

In [24]:

```
import warnings
warnings.filterwarnings("ignore")
start = datetime.now()

model3 = Sequential()
model3.add(Dense(450, activation='relu', input_shape=(input_dim,), kernel_initializer=
'he_normal'))
model3.add(Dense(200, activation='relu', kernel_initializer= 'he_normal'))
model3.add(Dense(90, activation='relu', kernel_initializer= 'he_normal'))
model3.add(Dense(output_dim, activation='softmax'))

print(model3.summary())

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))

print('Time taken :', datetime.now() - start)
```

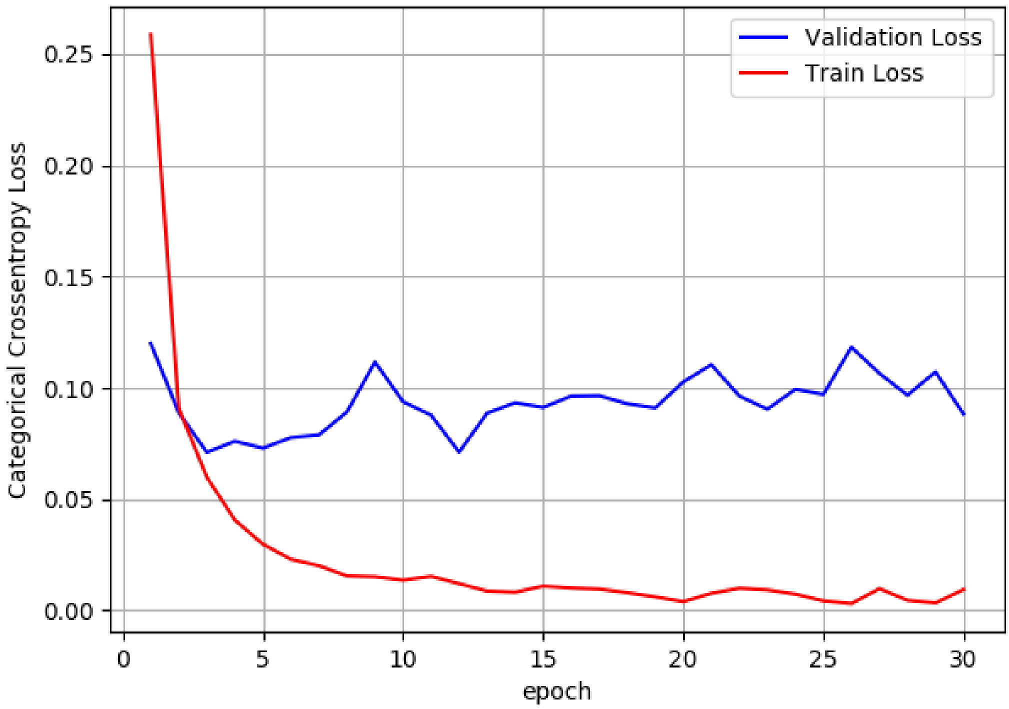
```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_15 (Dense)             (None, 450)               353250    
_________________________________________________________________
dense_16 (Dense)             (None, 200)               90200     
_________________________________________________________________
dense_17 (Dense)             (None, 90)                18090     
_________________________________________________________________
dense_18 (Dense)             (None, 10)                910       
=================================================================
Total params: 462,450
Trainable params: 462,450
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 9s 144us/step - loss: 0.258
8 - accuracy: 0.9239 - val_loss: 0.1199 - val_accuracy: 0.9639
Epoch 2/30
60000/60000 [==============================] - 7s 111us/step - loss: 0.090
7 - accuracy: 0.9724 - val_loss: 0.0888 - val_accuracy: 0.9722
Epoch 3/30
60000/60000 [==============================] - 7s 121us/step - loss: 0.059
8 - accuracy: 0.9811 - val_loss: 0.0710 - val_accuracy: 0.9777
Epoch 4/30
60000/60000 [==============================] - 7s 122us/step - loss: 0.040
6 - accuracy: 0.9878 - val_loss: 0.0760 - val_accuracy: 0.9776
Epoch 5/30
60000/60000 [==============================] - 8s 127us/step - loss: 0.029
7 - accuracy: 0.9902 - val_loss: 0.0729 - val_accuracy: 0.9777
Epoch 6/30
60000/60000 [==============================] - 7s 123us/step - loss: 0.023
0 - accuracy: 0.9926 - val_loss: 0.0777 - val_accuracy: 0.9782
Epoch 7/30
60000/60000 [==============================] - 7s 122us/step - loss: 0.020
1 - accuracy: 0.9933 - val_loss: 0.0789 - val_accuracy: 0.9789
Epoch 8/30
60000/60000 [==============================] - 7s 116us/step - loss: 0.015
5 - accuracy: 0.9949 - val_loss: 0.0892 - val_accuracy: 0.9766
Epoch 9/30
60000/60000 [==============================] - 7s 124us/step - loss: 0.015
1 - accuracy: 0.9950 - val_loss: 0.1116 - val_accuracy: 0.9732
Epoch 10/30
60000/60000 [==============================] - 7s 116us/step - loss: 0.013
6 - accuracy: 0.9953 - val_loss: 0.0938 - val_accuracy: 0.9760
Epoch 11/30
60000/60000 [==============================] - 7s 123us/step - loss: 0.015
3 - accuracy: 0.9948 - val_loss: 0.0877 - val_accuracy: 0.9774
Epoch 12/30
60000/60000 [==============================] - 7s 113us/step - loss: 0.012
0 - accuracy: 0.9959 - val_loss: 0.0711 - val_accuracy: 0.9825
Epoch 13/30
60000/60000 [==============================] - 7s 123us/step - loss: 0.008
7 - accuracy: 0.9973 - val_loss: 0.0886 - val_accuracy: 0.9799
Epoch 14/30
60000/60000 [==============================] - 7s 121us/step - loss: 0.008
2 - accuracy: 0.9969 - val_loss: 0.0933 - val_accuracy: 0.9801
Epoch 15/30
```

```
60000/60000 [==============================] - 7s 121us/step - loss: 0.010
9 - accuracy: 0.9962 - val_loss: 0.0912 - val_accuracy: 0.9790
Epoch 16/30
60000/60000 [==============================] - 7s 125us/step - loss: 0.010
1 - accuracy: 0.9964 - val_loss: 0.0963 - val_accuracy: 0.9803
Epoch 17/30
60000/60000 [==============================] - 7s 117us/step - loss: 0.009
6 - accuracy: 0.9967 - val_loss: 0.0965 - val_accuracy: 0.9808
Epoch 18/30
60000/60000 [==============================] - 7s 114us/step - loss: 0.008
0 - accuracy: 0.9974 - val_loss: 0.0929 - val_accuracy: 0.9803
Epoch 19/30
60000/60000 [==============================] - 8s 127us/step - loss: 0.006
1 - accuracy: 0.9983 - val_loss: 0.0910 - val_accuracy: 0.9816
Epoch 20/30
60000/60000 [==============================] - 7s 122us/step - loss: 0.003
9 - accuracy: 0.9989 - val_loss: 0.1026 - val_accuracy: 0.9813
Epoch 21/30
60000/60000 [==============================] - 7s 123us/step - loss: 0.007
7 - accuracy: 0.9976 - val_loss: 0.1104 - val_accuracy: 0.9766
Epoch 22/30
60000/60000 [==============================] - 8s 128us/step - loss: 0.010
0 - accuracy: 0.9969 - val_loss: 0.0963 - val_accuracy: 0.9808
Epoch 23/30
60000/60000 [==============================] - 8s 130us/step - loss: 0.009
2 - accuracy: 0.9969 - val_loss: 0.0904 - val_accuracy: 0.9812
Epoch 24/30
60000/60000 [==============================] - 8s 129us/step - loss: 0.007
3 - accuracy: 0.9978 - val_loss: 0.0994 - val_accuracy: 0.9793
Epoch 25/30
60000/60000 [==============================] - 8s 125us/step - loss: 0.004
3 - accuracy: 0.9987 - val_loss: 0.0971 - val_accuracy: 0.9809
Epoch 26/30
60000/60000 [==============================] - 8s 139us/step - loss: 0.003
1 - accuracy: 0.9991 - val_loss: 0.1183 - val_accuracy: 0.9784
Epoch 27/30
60000/60000 [==============================] - 8s 127us/step - loss: 0.009
8 - accuracy: 0.9968 - val_loss: 0.1065 - val_accuracy: 0.9797
Epoch 28/30
60000/60000 [==============================] - 7s 122us/step - loss: 0.004
5 - accuracy: 0.9983 - val_loss: 0.0967 - val_accuracy: 0.9814
Epoch 29/30
60000/60000 [==============================] - 8s 128us/step - loss: 0.003
3 - accuracy: 0.9989 - val_loss: 0.1072 - val_accuracy: 0.9805
Epoch 30/30
60000/60000 [==============================] - 8s 127us/step - loss: 0.009
4 - accuracy: 0.9970 - val_loss: 0.0883 - val_accuracy: 0.9821
Time taken : 0:03:44.069489
```

In [25]:

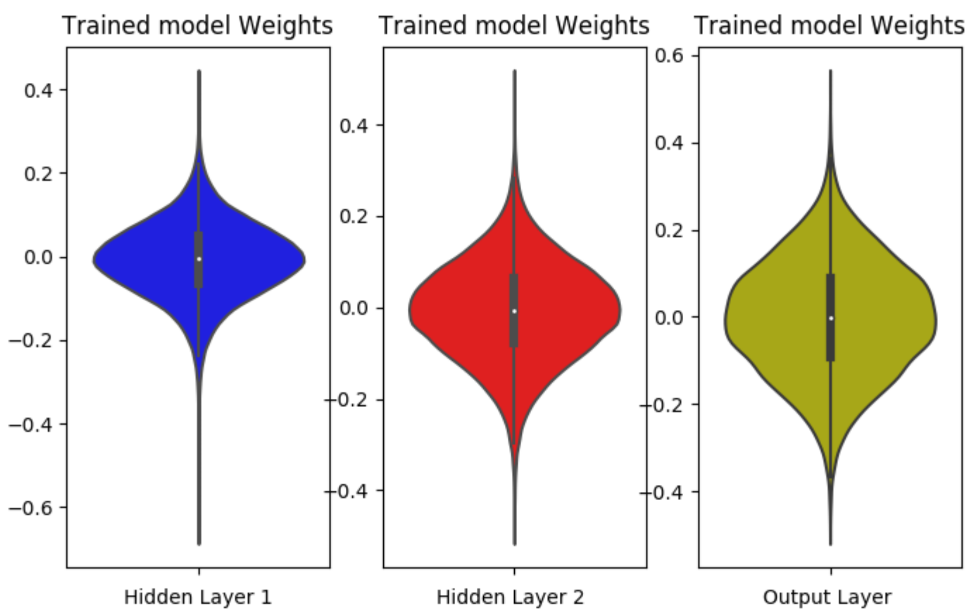
```
score3 = model3.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score3[0])
print('Test accuracy:', score3[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))
vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.08829341124333276
Test accuracy: 0.9821000099182129
```



In [26]:

```
w_after = model3.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



5.3.2 With Drop-out & Batch Normalization

In [27]:

```
import warnings
warnings.filterwarnings("ignore")
start = datetime.now()

model4 = Sequential()

model4.add(Dense(450, activation='relu', input_shape=(input_dim,), kernel_initializer=
'he_normal'))
model4.add(BatchNormalization())
model4.add(Dropout(0.5))

model4.add(Dense(200, activation='relu', kernel_initializer= 'he_normal'))
model4.add(BatchNormalization())
model4.add(Dropout(0.5))

model4.add(Dense(90, activation='relu', kernel_initializer= 'he_normal'))
model4.add(BatchNormalization())
model4.add(Dropout(0.5))

model4.add(Dense(output_dim, activation='softmax'))

print(model4.summary())

model4.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model4.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
1, validation_data=(X_test, Y_test))

print('Time taken :', datetime.now() - start)
```

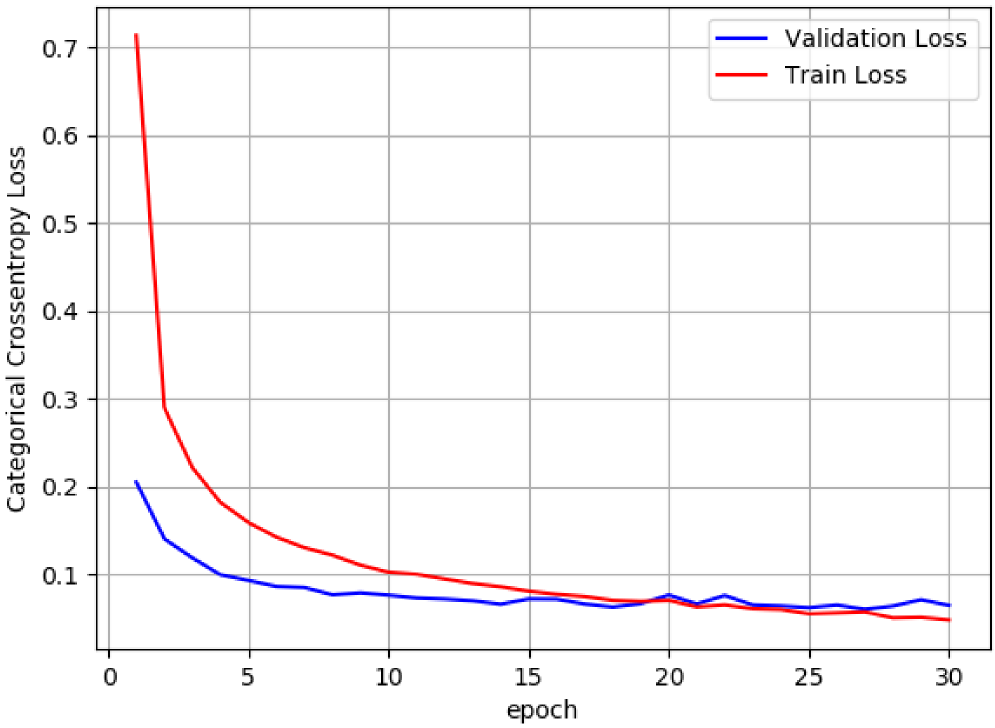
```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_19 (Dense)             (None, 450)               353250    
_________________________________________________________________
batch_normalization_4 (Batch (None, 450)               1800      
_________________________________________________________________
dropout_3 (Dropout)          (None, 450)               0         
_________________________________________________________________
dense_20 (Dense)             (None, 200)               90200     
_________________________________________________________________
batch_normalization_5 (Batch (None, 200)               800       
_________________________________________________________________
dropout_4 (Dropout)          (None, 200)               0         
_________________________________________________________________
dense_21 (Dense)             (None, 90)                18090     
_________________________________________________________________
batch_normalization_6 (Batch (None, 90)                360       
_________________________________________________________________
dropout_5 (Dropout)          (None, 90)                0         
_________________________________________________________________
dense_22 (Dense)             (None, 10)                910       
=================================================================
Total params: 465,410
Trainable params: 463,930
Non-trainable params: 1,480
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 15s 254us/step - loss: 0.71
32 - accuracy: 0.7818 - val_loss: 0.2050 - val_accuracy: 0.9390
Epoch 2/30
60000/60000 [==============================] - 12s 202us/step - loss: 0.28
98 - accuracy: 0.9158 - val_loss: 0.1404 - val_accuracy: 0.9568
Epoch 3/30
60000/60000 [==============================] - 12s 202us/step - loss: 0.22
14 - accuracy: 0.9354 - val_loss: 0.1188 - val_accuracy: 0.9636
Epoch 4/30
60000/60000 [==============================] - 12s 205us/step - loss: 0.18
19 - accuracy: 0.9466 - val_loss: 0.0996 - val_accuracy: 0.9690
Epoch 5/30
60000/60000 [==============================] - 12s 206us/step - loss: 0.15
91 - accuracy: 0.9540 - val_loss: 0.0930 - val_accuracy: 0.9725
Epoch 6/30
60000/60000 [==============================] - 12s 201us/step - loss: 0.14
23 - accuracy: 0.9592 - val_loss: 0.0862 - val_accuracy: 0.9727
Epoch 7/30
60000/60000 [==============================] - 12s 206us/step - loss: 0.13
03 - accuracy: 0.9620 - val_loss: 0.0850 - val_accuracy: 0.9749
Epoch 8/30
60000/60000 [==============================] - 12s 208us/step - loss: 0.12
19 - accuracy: 0.9645 - val_loss: 0.0767 - val_accuracy: 0.9771
Epoch 9/30
60000/60000 [==============================] - 12s 199us/step - loss: 0.11
05 - accuracy: 0.9673 - val_loss: 0.0790 - val_accuracy: 0.9772
Epoch 10/30
60000/60000 [==============================] - 12s 203us/step - loss: 0.10
25 - accuracy: 0.9694 - val_loss: 0.0763 - val_accuracy: 0.9785
Epoch 11/30
```

```
60000/60000 [==============================] - 12s 197us/step - loss: 0.10
02 - accuracy: 0.9707 - val_loss: 0.0732 - val_accuracy: 0.9799
Epoch 12/30
60000/60000 [==============================] - 12s 204us/step - loss: 0.09
47 - accuracy: 0.9721 - val_loss: 0.0720 - val_accuracy: 0.9796
Epoch 13/30
60000/60000 [==============================] - 12s 199us/step - loss: 0.08
95 - accuracy: 0.9738 - val_loss: 0.0700 - val_accuracy: 0.9812
Epoch 14/30
60000/60000 [==============================] - 12s 196us/step - loss: 0.08
59 - accuracy: 0.9749 - val_loss: 0.0661 - val_accuracy: 0.9810
Epoch 15/30
60000/60000 [==============================] - 12s 198us/step - loss: 0.08
08 - accuracy: 0.9762 - val_loss: 0.0721 - val_accuracy: 0.9803
Epoch 16/30
60000/60000 [==============================] - 12s 206us/step - loss: 0.07
73 - accuracy: 0.9765 - val_loss: 0.0718 - val_accuracy: 0.9807
Epoch 17/30
60000/60000 [==============================] - 12s 200us/step - loss: 0.07
48 - accuracy: 0.9778 - val_loss: 0.0663 - val_accuracy: 0.9816
Epoch 18/30
60000/60000 [==============================] - 12s 206us/step - loss: 0.07
05 - accuracy: 0.9788 - val_loss: 0.0629 - val_accuracy: 0.9825
Epoch 19/30
60000/60000 [==============================] - 12s 201us/step - loss: 0.06
94 - accuracy: 0.9796 - val_loss: 0.0667 - val_accuracy: 0.9812
Epoch 20/30
60000/60000 [==============================] - 12s 196us/step - loss: 0.07
05 - accuracy: 0.9787 - val_loss: 0.0765 - val_accuracy: 0.9799
Epoch 21/30
60000/60000 [==============================] - 12s 204us/step - loss: 0.06
31 - accuracy: 0.9811 - val_loss: 0.0665 - val_accuracy: 0.9826
Epoch 22/30
60000/60000 [==============================] - 13s 208us/step - loss: 0.06
53 - accuracy: 0.9805 - val_loss: 0.0760 - val_accuracy: 0.9799
Epoch 23/30
60000/60000 [==============================] - 12s 201us/step - loss: 0.06
09 - accuracy: 0.9823 - val_loss: 0.0652 - val_accuracy: 0.9832
Epoch 24/30
60000/60000 [==============================] - 12s 197us/step - loss: 0.06
01 - accuracy: 0.9818 - val_loss: 0.0641 - val_accuracy: 0.9833
Epoch 25/30
60000/60000 [==============================] - 12s 203us/step - loss: 0.05
51 - accuracy: 0.9834 - val_loss: 0.0622 - val_accuracy: 0.9838
Epoch 26/30
60000/60000 [==============================] - 12s 206us/step - loss: 0.05
61 - accuracy: 0.9827 - val_loss: 0.0651 - val_accuracy: 0.9826
Epoch 27/30
60000/60000 [==============================] - 12s 200us/step - loss: 0.05
72 - accuracy: 0.9829 - val_loss: 0.0604 - val_accuracy: 0.9827
Epoch 28/30
60000/60000 [==============================] - 12s 208us/step - loss: 0.05
04 - accuracy: 0.9844 - val_loss: 0.0640 - val_accuracy: 0.9834
Epoch 29/30
60000/60000 [==============================] - 12s 202us/step - loss: 0.05
10 - accuracy: 0.9842 - val_loss: 0.0712 - val_accuracy: 0.9824
Epoch 30/30
60000/60000 [==============================] - 12s 203us/step - loss: 0.04
82 - accuracy: 0.9849 - val_loss: 0.0648 - val_accuracy: 0.9834
Time taken : 0:06:09.783655
```

In [28]:

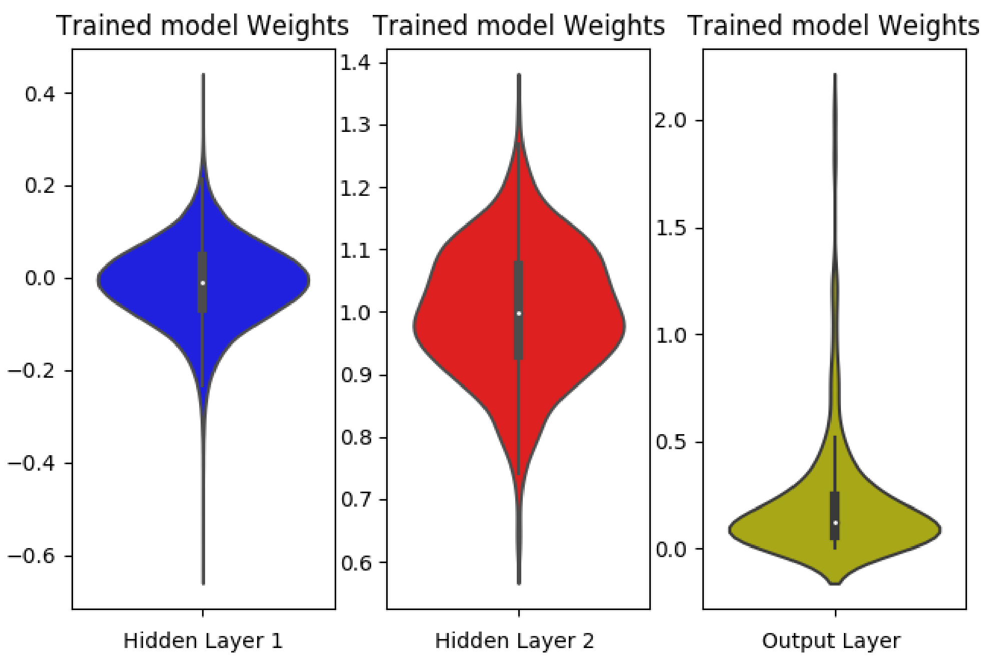
```
score4 = model4.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score4[0])
print('Test accuracy:', score4[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))
vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.06478801985616738
Test accuracy: 0.9833999872207642
```



In [29]:

```
w_after = model4.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



5.4 Five hidden layers with 784-450-300-200-150-90-10 architecture

5.4.1 Without Drop-out & Batch Normalization

In [30]:

```
import warnings
warnings.filterwarnings("ignore")
start = datetime.now()

model5 = Sequential()
model5.add(Dense(450, activation='relu', input_shape=(input_dim,), kernel_initializer=
'he_normal'))
model5.add(Dense(300, activation='relu', kernel_initializer= 'he_normal'))
model5.add(Dense(200, activation='relu', kernel_initializer= 'he_normal'))
model5.add(Dense(150, activation='relu', kernel_initializer= 'he_normal'))
model5.add(Dense(90, activation='relu', kernel_initializer= 'he_normal'))
model5.add(Dense(output_dim, activation='softmax'))

print(model5.summary())

model5.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model5.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
1, validation_data=(X_test, Y_test))

print('Time taken :', datetime.now() - start)
```

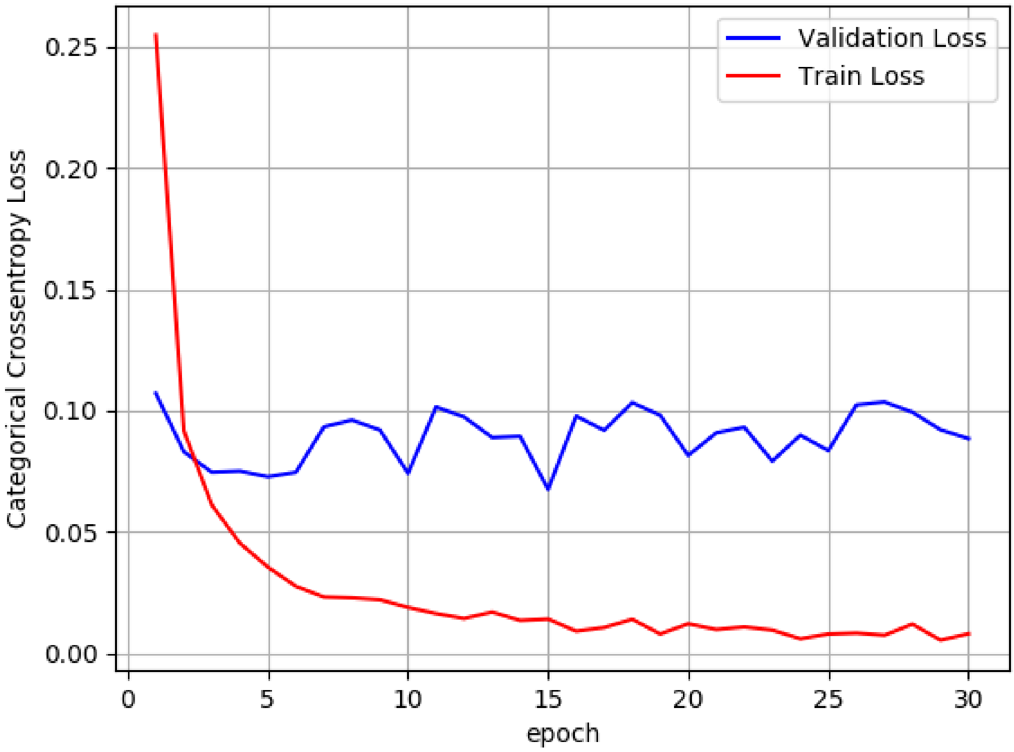
```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_23 (Dense)             (None, 450)               353250    
_________________________________________________________________
dense_24 (Dense)             (None, 300)               135300    
_________________________________________________________________
dense_25 (Dense)             (None, 200)               60200     
_________________________________________________________________
dense_26 (Dense)             (None, 150)               30150     
_________________________________________________________________
dense_27 (Dense)             (None, 90)                13590     
_________________________________________________________________
dense_28 (Dense)             (None, 10)                910       
=================================================================
Total params: 593,400
Trainable params: 593,400
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 10s 173us/step - loss: 0.25
48 - accuracy: 0.9237 - val_loss: 0.1071 - val_accuracy: 0.9677
Epoch 2/30
60000/60000 [==============================] - 9s 149us/step - loss: 0.091
7 - accuracy: 0.9717 - val_loss: 0.0830 - val_accuracy: 0.9737
Epoch 3/30
60000/60000 [==============================] - 10s 160us/step - loss: 0.06
10 - accuracy: 0.9811 - val_loss: 0.0746 - val_accuracy: 0.9760
Epoch 4/30
60000/60000 [==============================] - 9s 154us/step - loss: 0.045
3 - accuracy: 0.9856 - val_loss: 0.0750 - val_accuracy: 0.9792
Epoch 5/30
60000/60000 [==============================] - 10s 158us/step - loss: 0.03
55 - accuracy: 0.9887 - val_loss: 0.0727 - val_accuracy: 0.9797
Epoch 6/30
60000/60000 [==============================] - 10s 161us/step - loss: 0.02
75 - accuracy: 0.9910 - val_loss: 0.0745 - val_accuracy: 0.9789
Epoch 7/30
60000/60000 [==============================] - 9s 156us/step - loss: 0.023
1 - accuracy: 0.9924 - val_loss: 0.0933 - val_accuracy: 0.9763
Epoch 8/30
60000/60000 [==============================] - 9s 157us/step - loss: 0.022
8 - accuracy: 0.9927 - val_loss: 0.0961 - val_accuracy: 0.9749
Epoch 9/30
60000/60000 [==============================] - 9s 158us/step - loss: 0.022
0 - accuracy: 0.9920 - val_loss: 0.0920 - val_accuracy: 0.9770
Epoch 10/30
60000/60000 [==============================] - 9s 156us/step - loss: 0.018
8 - accuracy: 0.9940 - val_loss: 0.0741 - val_accuracy: 0.9814
Epoch 11/30
60000/60000 [==============================] - 9s 155us/step - loss: 0.016
3 - accuracy: 0.9948 - val_loss: 0.1015 - val_accuracy: 0.9748
Epoch 12/30
60000/60000 [==============================] - 9s 150us/step - loss: 0.014
3 - accuracy: 0.9956 - val_loss: 0.0974 - val_accuracy: 0.9777
Epoch 13/30
60000/60000 [==============================] - 9s 153us/step - loss: 0.016
9 - accuracy: 0.9947 - val_loss: 0.0889 - val_accuracy: 0.9810
```

```
Epoch 14/30
60000/60000 [==============================] - 10s 159us/step - loss: 0.01
35 - accuracy: 0.9957 - val_loss: 0.0894 - val_accuracy: 0.9790
Epoch 15/30
60000/60000 [==============================] - 9s 156us/step - loss: 0.014
0 - accuracy: 0.9955 - val_loss: 0.0675 - val_accuracy: 0.9837
Epoch 16/30
60000/60000 [==============================] - 9s 150us/step - loss: 0.009
0 - accuracy: 0.9972 - val_loss: 0.0977 - val_accuracy: 0.9792
Epoch 17/30
60000/60000 [==============================] - 9s 154us/step - loss: 0.010
6 - accuracy: 0.9968 - val_loss: 0.0918 - val_accuracy: 0.9818
Epoch 18/30
60000/60000 [==============================] - 9s 158us/step - loss: 0.013
9 - accuracy: 0.9960 - val_loss: 0.1033 - val_accuracy: 0.9759
Epoch 19/30
60000/60000 [==============================] - 9s 154us/step - loss: 0.007
9 - accuracy: 0.9975 - val_loss: 0.0980 - val_accuracy: 0.9793
Epoch 20/30
60000/60000 [==============================] - 9s 151us/step - loss: 0.012
1 - accuracy: 0.9963 - val_loss: 0.0815 - val_accuracy: 0.9805
Epoch 21/30
60000/60000 [==============================] - 9s 157us/step - loss: 0.009
7 - accuracy: 0.9971 - val_loss: 0.0908 - val_accuracy: 0.9824
Epoch 22/30
60000/60000 [==============================] - 9s 145us/step - loss: 0.010
8 - accuracy: 0.9966 - val_loss: 0.0931 - val_accuracy: 0.9824
Epoch 23/30
60000/60000 [==============================] - 9s 154us/step - loss: 0.009
5 - accuracy: 0.9972 - val_loss: 0.0792 - val_accuracy: 0.9826
Epoch 24/30
60000/60000 [==============================] - 10s 159us/step - loss: 0.00
59 - accuracy: 0.9984 - val_loss: 0.0898 - val_accuracy: 0.9820
Epoch 25/30
60000/60000 [==============================] - 9s 156us/step - loss: 0.007
9 - accuracy: 0.9979 - val_loss: 0.0834 - val_accuracy: 0.9827
Epoch 26/30
60000/60000 [==============================] - 9s 155us/step - loss: 0.008
3 - accuracy: 0.9973 - val_loss: 0.1023 - val_accuracy: 0.9785
Epoch 27/30
60000/60000 [==============================] - 9s 157us/step - loss: 0.007
4 - accuracy: 0.9979 - val_loss: 0.1035 - val_accuracy: 0.9775
Epoch 28/30
60000/60000 [==============================] - 10s 166us/step - loss: 0.01
19 - accuracy: 0.9967 - val_loss: 0.0993 - val_accuracy: 0.9802
Epoch 29/30
60000/60000 [==============================] - 10s 162us/step - loss: 0.00
53 - accuracy: 0.9984 - val_loss: 0.0920 - val_accuracy: 0.9821
Epoch 30/30
60000/60000 [==============================] - 10s 160us/step - loss: 0.00
80 - accuracy: 0.9975 - val_loss: 0.0884 - val_accuracy: 0.9811
Time taken : 0:04:43.392793
```

In [31]:

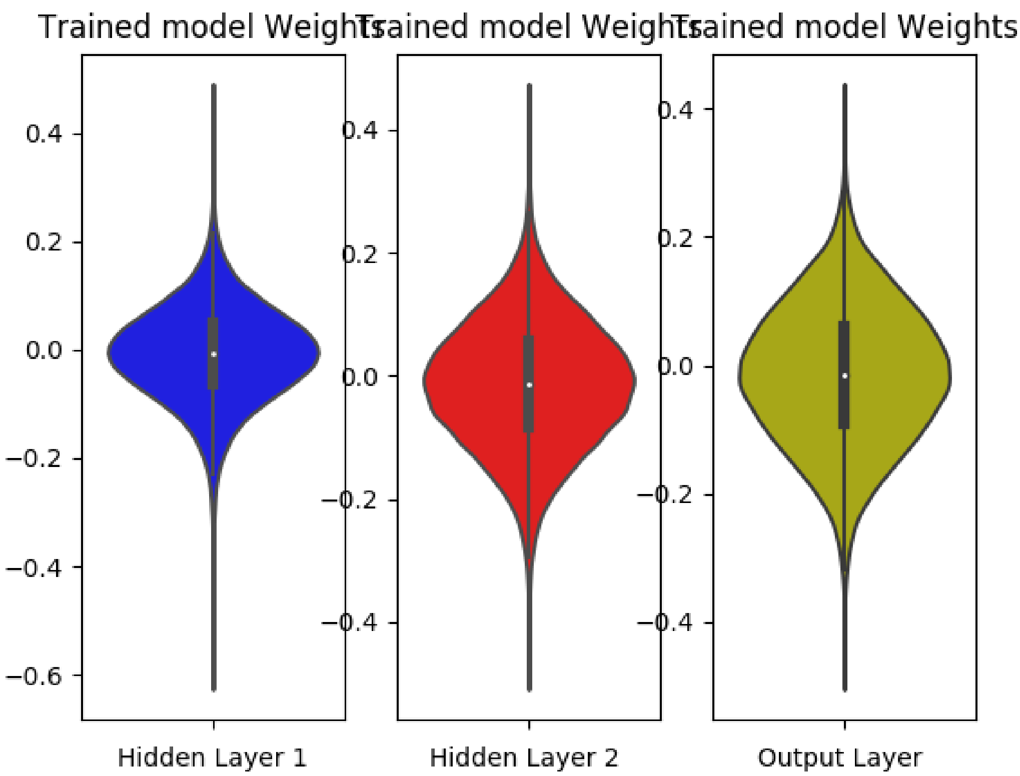
```
score5 = model5.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score5[0])
print('Test accuracy:', score5[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

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

```
Test score: 0.08842633471704 16
Test accuracy: 0.9811000227928162
```



In [32]:

```
w_after = model5.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



5.4.2 With Drop-out & Batch Normalization

In [33]:

```
import warnings
warnings.filterwarnings("ignore")
start = datetime.now()

model6 = Sequential()

model6.add(Dense(450, activation='relu', input_shape=(input_dim,), kernel_initializer=
'he_normal'))
model6.add(BatchNormalization())
model6.add(Dropout(0.5))

model6.add(Dense(300, activation='relu', kernel_initializer= 'he_normal'))
model6.add(BatchNormalization())
model6.add(Dropout(0.5))

model6.add(Dense(200, activation='relu', kernel_initializer= 'he_normal'))
model6.add(BatchNormalization())
model6.add(Dropout(0.5))

model6.add(Dense(150, activation='relu', kernel_initializer= 'he_normal'))
model6.add(BatchNormalization())
model6.add(Dropout(0.5))

model6.add(Dense(90, activation='relu', kernel_initializer= 'he_normal'))
model6.add(BatchNormalization())
model6.add(Dropout(0.5))

model6.add(Dense(output_dim, activation='softmax'))

print(model6.summary())

model6.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model6.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
1, validation_data=(X_test, Y_test))

print('Time taken :', datetime.now() - start)
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_29 (Dense)             (None, 450)               353250    
_________________________________________________________________
batch_normalization_7 (Batch (None, 450)               1800      
_________________________________________________________________
dropout_6 (Dropout)          (None, 450)               0         
_________________________________________________________________
dense_30 (Dense)             (None, 300)               135300    
_________________________________________________________________
batch_normalization_8 (Batch (None, 300)               1200      
_________________________________________________________________
dropout_7 (Dropout)          (None, 300)               0         
_________________________________________________________________
dense_31 (Dense)             (None, 200)               60200     
_________________________________________________________________
batch_normalization_9 (Batch (None, 200)               800       
_________________________________________________________________
dropout_8 (Dropout)          (None, 200)               0         
_________________________________________________________________
dense_32 (Dense)             (None, 150)               30150     
_________________________________________________________________
batch_normalization_10 (Batc (None, 150)               600       
_________________________________________________________________
dropout_9 (Dropout)          (None, 150)               0         
_________________________________________________________________
dense_33 (Dense)             (None, 90)                13590     
_________________________________________________________________
batch_normalization_11 (Batc (None, 90)                360       
_________________________________________________________________
dropout_10 (Dropout)         (None, 90)                0         
_________________________________________________________________
dense_34 (Dense)             (None, 10)                910       
=================================================================
Total params: 598,160
Trainable params: 595,780
Non-trainable params: 2,380
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 23s 379us/step - loss: 1.33
28 - accuracy: 0.5824 - val_loss: 0.3538 - val_accuracy: 0.8937
Epoch 2/30
60000/60000 [==============================] - 19s 311us/step - loss: 0.44
88 - accuracy: 0.8709 - val_loss: 0.1935 - val_accuracy: 0.9436
Epoch 3/30
60000/60000 [==============================] - 18s 300us/step - loss: 0.31
56 - accuracy: 0.9134 - val_loss: 0.1593 - val_accuracy: 0.9554
Epoch 4/30
60000/60000 [==============================] - 18s 306us/step - loss: 0.25
85 - accuracy: 0.9303 - val_loss: 0.1410 - val_accuracy: 0.9619
Epoch 5/30
60000/60000 [==============================] - 19s 309us/step - loss: 0.21
53 - accuracy: 0.9427 - val_loss: 0.1274 - val_accuracy: 0.9667
Epoch 6/30
60000/60000 [==============================] - 18s 294us/step - loss: 0.19
54 - accuracy: 0.9487 - val_loss: 0.1095 - val_accuracy: 0.9716
Epoch 7/30
```

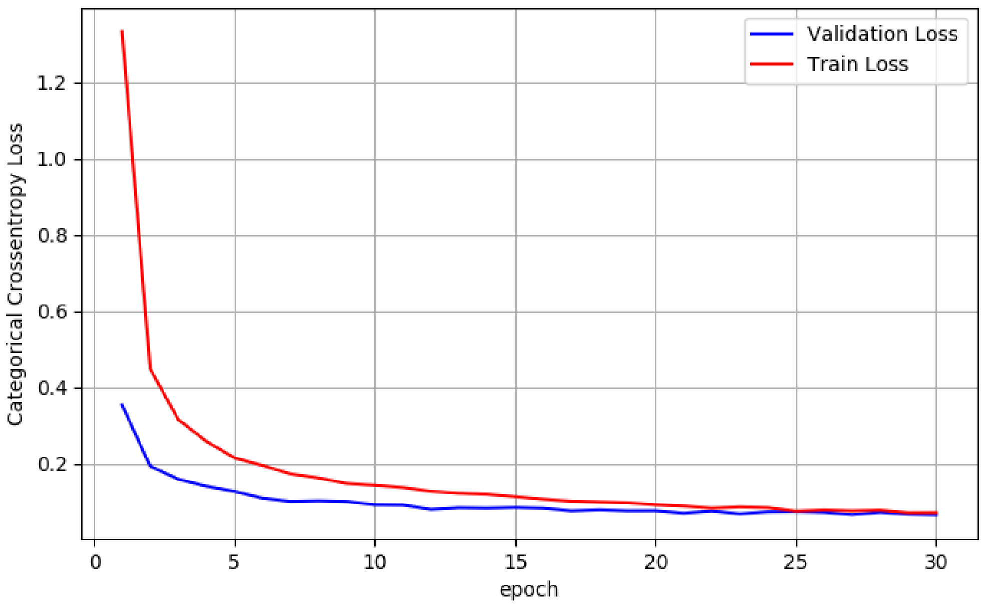
```
60000/60000 [==============================] - 18s 301us/step - loss: 0.17
31 - accuracy: 0.9539 - val_loss: 0.1009 - val_accuracy: 0.9739
Epoch 8/30
60000/60000 [==============================] - 18s 301us/step - loss: 0.16
25 - accuracy: 0.9574 - val_loss: 0.1026 - val_accuracy: 0.9730
Epoch 9/30
60000/60000 [==============================] - 18s 297us/step - loss: 0.14
82 - accuracy: 0.9612 - val_loss: 0.1006 - val_accuracy: 0.9738
Epoch 10/30
60000/60000 [==============================] - 18s 303us/step - loss: 0.14
36 - accuracy: 0.9621 - val_loss: 0.0925 - val_accuracy: 0.9761
Epoch 11/30
60000/60000 [==============================] - 18s 302us/step - loss: 0.13
76 - accuracy: 0.9634 - val_loss: 0.0919 - val_accuracy: 0.9764
Epoch 12/30
60000/60000 [==============================] - 18s 297us/step - loss: 0.12
75 - accuracy: 0.9661 - val_loss: 0.0802 - val_accuracy: 0.9786
Epoch 13/30
60000/60000 [==============================] - 17s 290us/step - loss: 0.12
25 - accuracy: 0.9668 - val_loss: 0.0847 - val_accuracy: 0.9774
Epoch 14/30
60000/60000 [==============================] - 17s 289us/step - loss: 0.11
98 - accuracy: 0.9682 - val_loss: 0.0837 - val_accuracy: 0.9787
Epoch 15/30
60000/60000 [==============================] - 18s 295us/step - loss: 0.11
35 - accuracy: 0.9698 - val_loss: 0.0853 - val_accuracy: 0.9788
Epoch 16/30
60000/60000 [==============================] - 18s 302us/step - loss: 0.10
68 - accuracy: 0.9711 - val_loss: 0.0835 - val_accuracy: 0.9782
Epoch 17/30
60000/60000 [==============================] - 18s 307us/step - loss: 0.10
15 - accuracy: 0.9728 - val_loss: 0.0765 - val_accuracy: 0.9815
Epoch 18/30
60000/60000 [==============================] - 18s 298us/step - loss: 0.09
94 - accuracy: 0.9736 - val_loss: 0.0791 - val_accuracy: 0.9790
Epoch 19/30
60000/60000 [==============================] - 18s 293us/step - loss: 0.09
75 - accuracy: 0.9744 - val_loss: 0.0767 - val_accuracy: 0.9818
Epoch 20/30
60000/60000 [==============================] - 18s 306us/step - loss: 0.09
25 - accuracy: 0.9750 - val_loss: 0.0768 - val_accuracy: 0.9801
Epoch 21/30
60000/60000 [==============================] - 18s 302us/step - loss: 0.08
89 - accuracy: 0.9768 - val_loss: 0.0702 - val_accuracy: 0.9823
Epoch 22/30
60000/60000 [==============================] - 18s 302us/step - loss: 0.08
38 - accuracy: 0.9772 - val_loss: 0.0761 - val_accuracy: 0.9811
Epoch 23/30
60000/60000 [==============================] - 18s 295us/step - loss: 0.08
67 - accuracy: 0.9776 - val_loss: 0.0685 - val_accuracy: 0.9835
Epoch 24/30
60000/60000 [==============================] - 17s 289us/step - loss: 0.08
49 - accuracy: 0.9775 - val_loss: 0.0739 - val_accuracy: 0.9819
Epoch 25/30
60000/60000 [==============================] - 18s 302us/step - loss: 0.07
57 - accuracy: 0.9789 - val_loss: 0.0746 - val_accuracy: 0.9810
Epoch 26/30
60000/60000 [==============================] - 18s 300us/step - loss: 0.07
87 - accuracy: 0.9790 - val_loss: 0.0725 - val_accuracy: 0.9829
Epoch 27/30
60000/60000 [==============================] - 17s 289us/step - loss: 0.07
```

```
66 - accuracy: 0.9791 - val_loss: 0.0664 - val_accuracy: 0.9841
Epoch 28/30
60000/60000 [==============================] - 18s 297us/step - loss: 0.07
83 - accuracy: 0.9794 - val_loss: 0.0725 - val_accuracy: 0.9823
Epoch 29/30
60000/60000 [==============================] - 18s 297us/step - loss: 0.07
15 - accuracy: 0.9807 - val_loss: 0.0672 - val_accuracy: 0.9829
Epoch 30/30
60000/60000 [==============================] - 18s 292us/step - loss: 0.07
18 - accuracy: 0.9801 - val_loss: 0.0658 - val_accuracy: 0.9830
Time taken : 0:09:07.031936
```

In [34]:

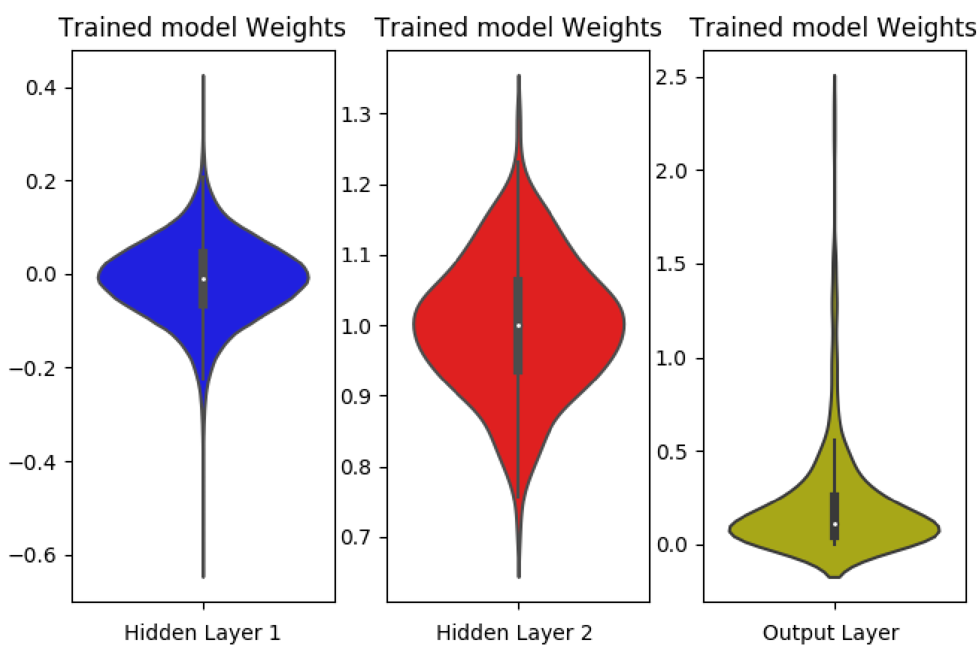
```
score6 = model6.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score6[0])
print('Test accuracy:', score6[1])

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

# list of epoch numbers
x = list(range(1,nb_epoch+1))

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

```
Test score: 0.0658454667896498
Test accuracy: 0.9829999804496765
```



In [35]:

```
w_after = model6.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained 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()
```



5.5 Summary

In [28]:

```
import pandas as pd

df= pd.DataFrame(columns=["Number of Layers","Activation function count","Test Loss(without drop-out & batch normalization)","Test Accuracy (without drop-out & batch normalization)","Test Loss (with drop-out & batch normalization)","Test Accuracy (with drop-out & batch normalization)"],index=['I','II','III'])
df.loc['I']=[2,(400,80),0.087, 0.9848, 0.0637, 0.9825]
df.loc['II']=[3,(450,200,90),0.08882, 0.9821, 0.0648, 0.9833]
df.loc['III']=[5,(450,300,200,150,90),0.0884, 0.9811, 0.0658, 0.9830]
df
```

Out[28]:

	Number of Layers	Activation function count	Test Loss(without drop-out & batch normalization)	Test Accuracy (without drop-out & batch normalization)	Test Loss (with drop-out & batch normalization)	Test Accuracy (with drop-out & batch normalization)
I	2	(400, 80)	0.087	0.9848	0.0637	0.9825
II	3	(450, 200, 90)	0.08882	0.9821	0.0648	0.9833
III	5	(450, 300, 200, 150, 90)	0.0884	0.9811	0.0658	0.983

- From the above table it can be observed that the accuracy for all the models (except for the 1st model) was marginally high when drop-out & batch normalization was used
- Also test loss was significantly less in drop-out & batch normalization models