Keras -- MLPs on MNIST

In [1]:

```
# if you keras is not using tensorflow as backend set "KERAS_BACKEND=tensorflow" use this c
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()
```

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 (http
s://s3.amazonaws.com/img-datasets/mnist.npz)
11493376/11490434 [==============================] - 10s 1us/step
```

In [4]:

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

```
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 3 dimensional vector
# for each image we have a (28*28) vector
# we will convert the (28*28) vector into single dimensional vector of 1 * 784

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

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

```
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.
```

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.]
```

Softmax classifier

In [11]:

```
# https://keras.io/getting-started/sequential-model-guide/

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

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

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

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

###

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

# keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_unif
# bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regula
# 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 arg

# 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 [12]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [13]:

```
print(X_train.shape[1])
```

```
784
```

MLP + ReLU + ADAM with 2 layers without Dropout and Batch Normalisation

In [14]:

```
model_relu = Sequential()
model_relu.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_relu.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

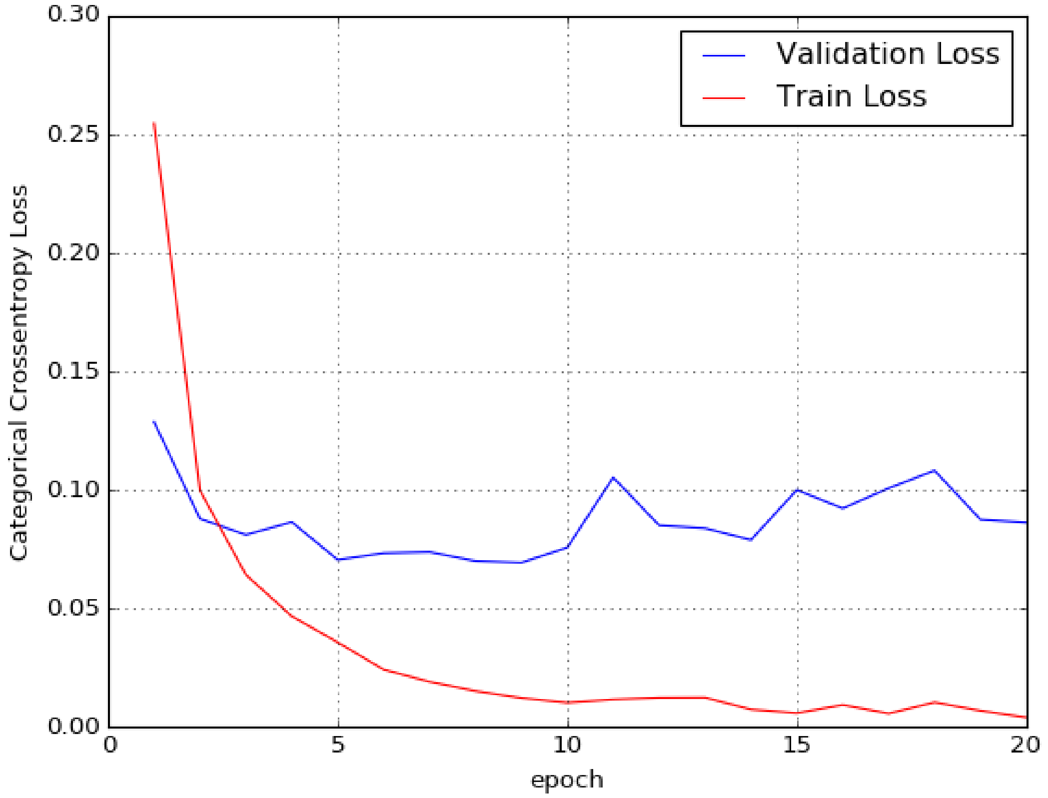
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 364)               285740    
_________________________________________________________________
dense_2 (Dense)              (None, 52)                18980     
_________________________________________________________________
dense_3 (Dense)              (None, 10)                530       
=================================================================
Total params: 305,250
Trainable params: 305,250
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.2546 -
acc: 0.9248 - val_loss: 0.1287 - val_acc: 0.9603
Epoch 2/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.0997 -
acc: 0.9703 - val_loss: 0.0878 - val_acc: 0.9728
Epoch 3/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0641 -
acc: 0.9806 - val_loss: 0.0810 - val_acc: 0.9757
Epoch 4/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0466 -
acc: 0.9857 - val_loss: 0.0864 - val_acc: 0.9748
Epoch 5/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0356 -
acc: 0.9884 - val_loss: 0.0704 - val_acc: 0.9781
Epoch 6/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0241 -
acc: 0.9930 - val_loss: 0.0732 - val_acc: 0.9783
Epoch 7/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0189 -
acc: 0.9943 - val_loss: 0.0737 - val_acc: 0.9782
Epoch 8/20
60000/60000 [==============================] - 8s 125us/step - loss: 0.0150
- acc: 0.9952 - val_loss: 0.0699 - val_acc: 0.9809
Epoch 9/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0121
- acc: 0.9963 - val_loss: 0.0692 - val_acc: 0.9797
Epoch 10/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0101
- acc: 0.9969 - val_loss: 0.0756 - val_acc: 0.9805
Epoch 11/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0115
```

```
- acc: 0.9963 - val_loss: 0.1052 - val_acc: 0.9741
Epoch 12/20
60000/60000 [==============================] - 7s 117us/step - loss: 0.0122
- acc: 0.9961 - val_loss: 0.0850 - val_acc: 0.9799
Epoch 13/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.0122
- acc: 0.9958 - val_loss: 0.0838 - val_acc: 0.9796
Epoch 14/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0072
- acc: 0.9978 - val_loss: 0.0788 - val_acc: 0.9816
Epoch 15/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0057 -
acc: 0.9983 - val_loss: 0.0999 - val_acc: 0.9783
Epoch 16/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0091 -
acc: 0.9970 - val_loss: 0.0921 - val_acc: 0.9797
Epoch 17/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0055 -
acc: 0.9982 - val_loss: 0.1007 - val_acc: 0.9792
Epoch 18/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0102 -
acc: 0.9968 - val_loss: 0.1081 - val_acc: 0.9788
Epoch 19/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0067 -
acc: 0.9978 - val_loss: 0.0874 - val_acc: 0.9807
Epoch 20/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.0039 -
acc: 0.9989 - val_loss: 0.0861 - val_acc: 0.9821
```

In [15]:

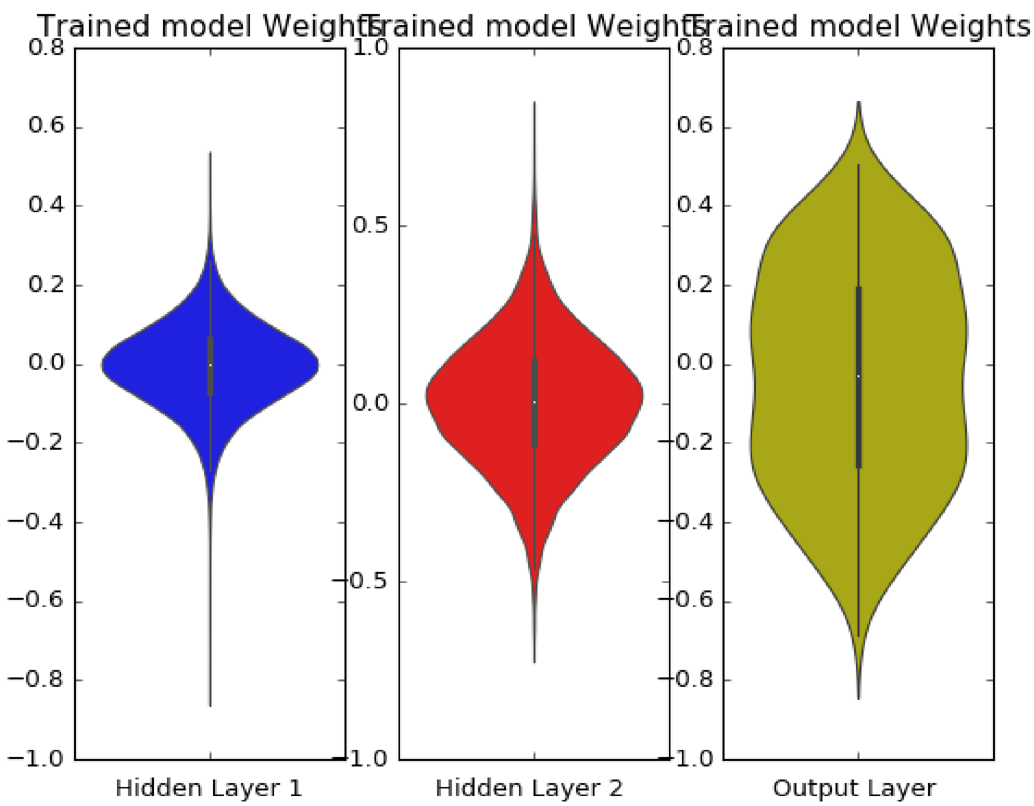
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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


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

```
Test score: 0.08614989673080777
Test accuracy: 0.9821
```



In [16]:

```
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()
```



MLP + Batch-Norm on 2 hidden Layers + AdamOptimizer

In [17]:

```
# 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=v
# 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(364, activation='relu', input_shape=(input_dim,), kernel_initializer=
model_batch.add(BatchNormalization())

model_batch.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_4 (Dense)              (None, 364)               285740    
_________________________________________________________________
batch_normalization_1 (Batch (None, 364)               1456      
_________________________________________________________________
dense_5 (Dense)              (None, 52)                18980     
_________________________________________________________________
batch_normalization_2 (Batch (None, 52)                208       
_________________________________________________________________
dense_6 (Dense)              (None, 10)                530       
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

In [18]:

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

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose
```

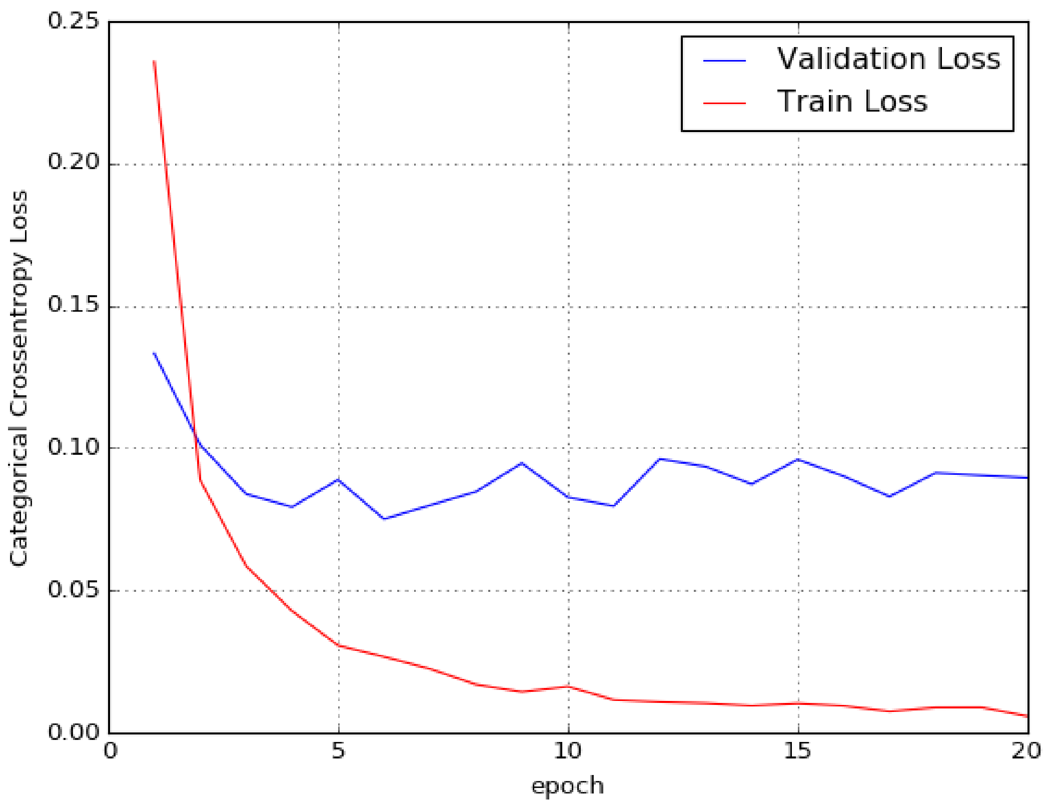
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.2357
- acc: 0.9333 - val_loss: 0.1332 - val_acc: 0.9596
Epoch 2/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0887 -
acc: 0.9737 - val_loss: 0.1011 - val_acc: 0.9693
Epoch 3/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0585 -
acc: 0.9826 - val_loss: 0.0838 - val_acc: 0.9760
Epoch 4/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0427 -
acc: 0.9871 - val_loss: 0.0793 - val_acc: 0.9749
Epoch 5/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0306 -
acc: 0.9904 - val_loss: 0.0888 - val_acc: 0.9748
Epoch 6/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.0266 -
acc: 0.9915 - val_loss: 0.0750 - val_acc: 0.9794
Epoch 7/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0224 -
acc: 0.9931 - val_loss: 0.0798 - val_acc: 0.9771
Epoch 8/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.0169 -
acc: 0.9952 - val_loss: 0.0846 - val_acc: 0.9746
Epoch 9/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.0143 -
acc: 0.9957 - val_loss: 0.0946 - val_acc: 0.9752
Epoch 10/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0162 -
acc: 0.9947 - val_loss: 0.0827 - val_acc: 0.9780
Epoch 11/20
60000/60000 [==============================] - 9s 149us/step - loss: 0.0115
- acc: 0.9962 - val_loss: 0.0796 - val_acc: 0.9778
Epoch 12/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0108
- acc: 0.9969 - val_loss: 0.0961 - val_acc: 0.9764
Epoch 13/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0103 -
acc: 0.9967 - val_loss: 0.0934 - val_acc: 0.9760
Epoch 14/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0094
- acc: 0.9972 - val_loss: 0.0873 - val_acc: 0.9788
Epoch 15/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.0103
- acc: 0.9966 - val_loss: 0.0959 - val_acc: 0.9762
Epoch 16/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.0094
- acc: 0.9970 - val_loss: 0.0901 - val_acc: 0.9789
Epoch 17/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0074
- acc: 0.9977 - val_loss: 0.0829 - val_acc: 0.9795
Epoch 18/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0088 -
acc: 0.9971 - val_loss: 0.0912 - val_acc: 0.9773
```

```
Epoch 19/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0088 -
acc: 0.9970 - val_loss: 0.0904 - val_acc: 0.9776
Epoch 20/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0058 -
acc: 0.9981 - val_loss: 0.0895 - val_acc: 0.9792
```

In [20]:

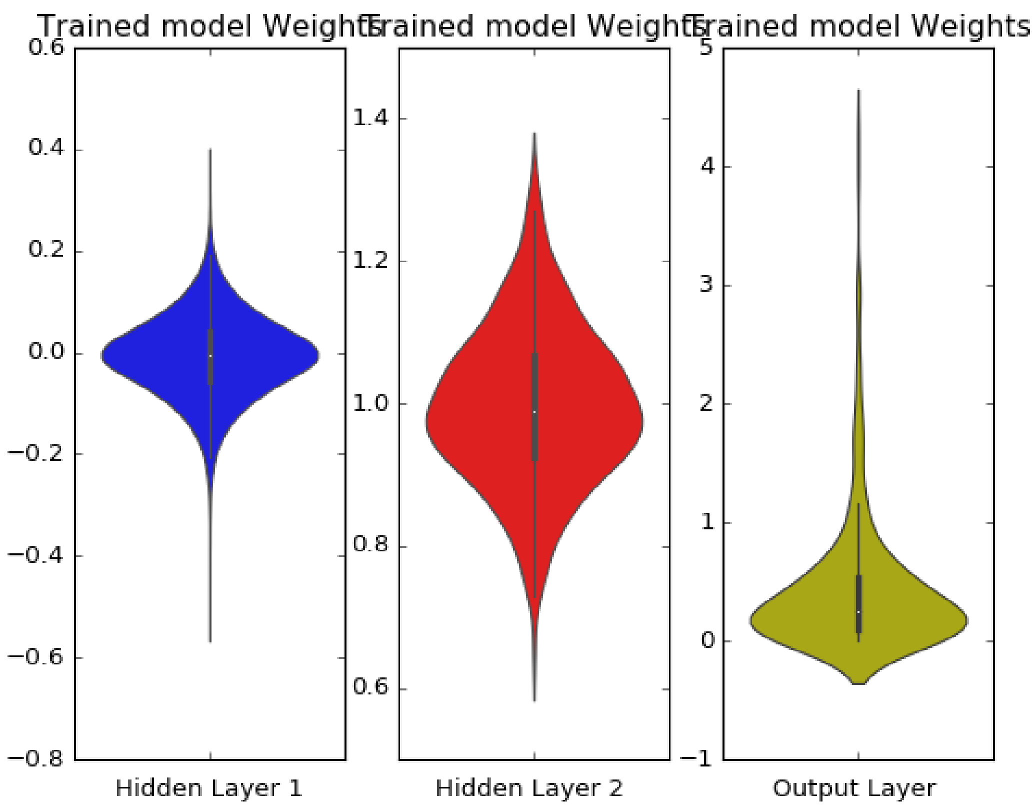
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.08952085421274897
Test accuracy: 0.9792
```



In [21]:

```
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()
```



MLP + Dropout (rate = 0.5) + AdamOptimizer with 2 hidden layers

In [22]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(Dropout(0.5))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_7 (Dense)              (None, 364)               285740    
_________________________________________________________________
dropout_1 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_8 (Dense)              (None, 52)                18980     
_________________________________________________________________
dropout_2 (Dropout)          (None, 52)                0         
_________________________________________________________________
dense_9 (Dense)              (None, 10)                530       
=================================================================
Total params: 305,250
Trainable params: 305,250
Non-trainable params: 0
_________________________________________________________________
```

In [23]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

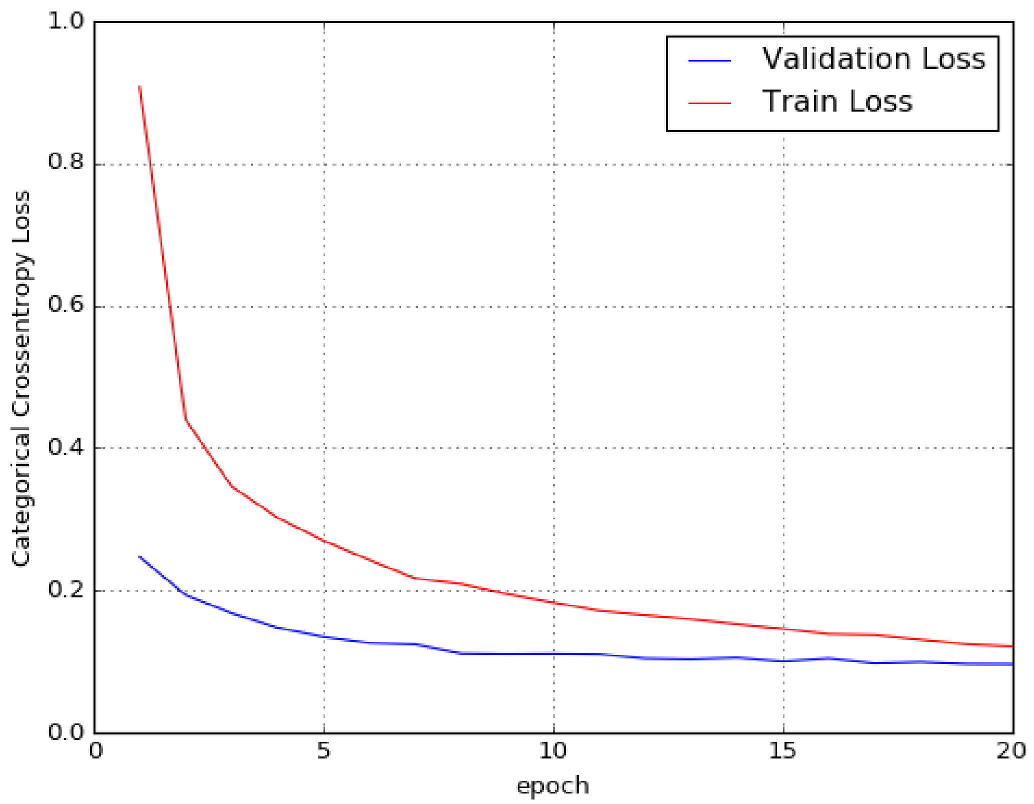
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.9070 -
acc: 0.7097 - val_loss: 0.2468 - val_acc: 0.9309
Epoch 2/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.4398 -
acc: 0.8680 - val_loss: 0.1933 - val_acc: 0.9446
Epoch 3/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.3459 -
acc: 0.8991 - val_loss: 0.1679 - val_acc: 0.9530
Epoch 4/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.3020 -
acc: 0.9151 - val_loss: 0.1473 - val_acc: 0.9577
Epoch 5/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.2697 -
acc: 0.9248 - val_loss: 0.1346 - val_acc: 0.9635
Epoch 6/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.2425 -
acc: 0.9330 - val_loss: 0.1258 - val_acc: 0.9637
Epoch 7/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.2166 -
acc: 0.9393 - val_loss: 0.1239 - val_acc: 0.9664
Epoch 8/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.2087 -
acc: 0.9416 - val_loss: 0.1113 - val_acc: 0.9706
Epoch 9/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.1947 -
acc: 0.9464 - val_loss: 0.1106 - val_acc: 0.9709
Epoch 10/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.1828 -
acc: 0.9490 - val_loss: 0.1110 - val_acc: 0.9702
Epoch 11/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.1713 -
acc: 0.9524 - val_loss: 0.1102 - val_acc: 0.9718
Epoch 12/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.1651
- acc: 0.9533 - val_loss: 0.1041 - val_acc: 0.9725
Epoch 13/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.1594 -
acc: 0.9558 - val_loss: 0.1029 - val_acc: 0.9735
Epoch 14/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1524 -
acc: 0.9574 - val_loss: 0.1051 - val_acc: 0.9737
Epoch 15/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.1457
- acc: 0.9592 - val_loss: 0.1000 - val_acc: 0.9746
Epoch 16/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.1384
- acc: 0.9602 - val_loss: 0.1041 - val_acc: 0.9763
Epoch 17/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.1372 -
acc: 0.9613 - val_loss: 0.0975 - val_acc: 0.9759
Epoch 18/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.1306 -
acc: 0.9638 - val_loss: 0.0993 - val_acc: 0.9753
```

```
Epoch 19/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1241 -
acc: 0.9656 - val_loss: 0.0965 - val_acc: 0.9763
Epoch 20/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.1210
- acc: 0.9660 - val_loss: 0.0963 - val_acc: 0.9756
```

In [24]:

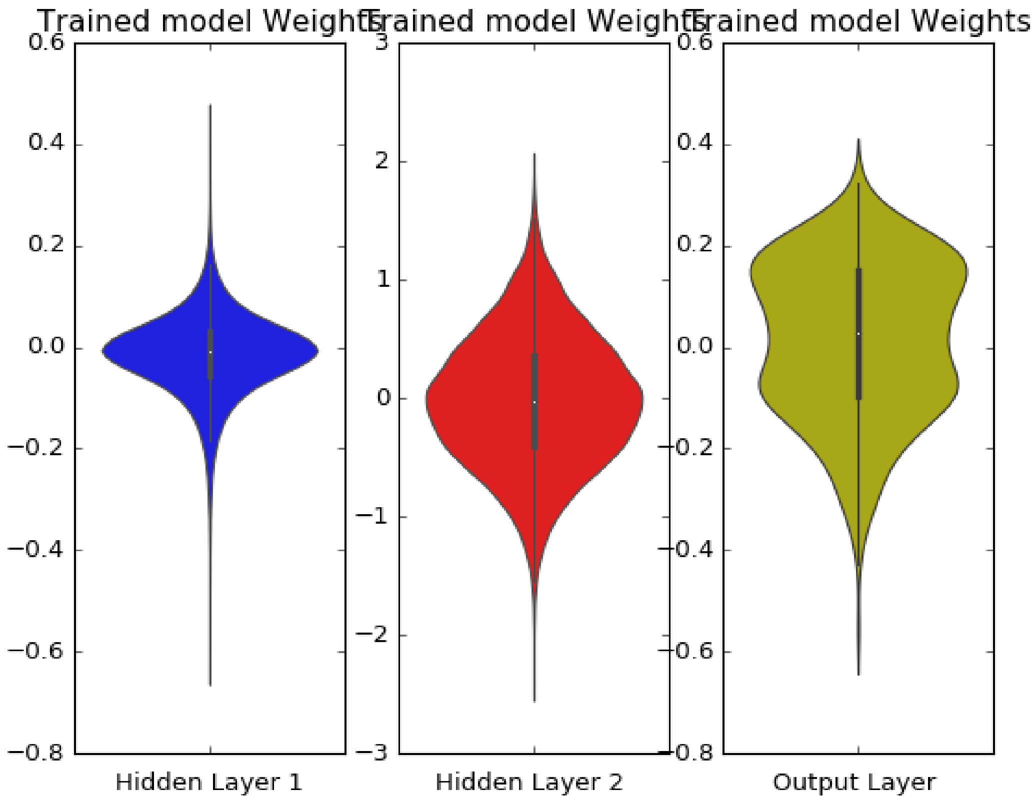
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.09625575973058877
Test accuracy: 0.9756
```



In [25]:

```
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()
```



In [ ]:

```
```

MLP + Dropout (rate = 0.3) + AdamOptimizer with 2 hidden layers

In [26]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(Dropout(0.3))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.3))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_10 (Dense)             (None, 364)               285740    
_________________________________________________________________
dropout_3 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 52)                18980     
_________________________________________________________________
dropout_4 (Dropout)          (None, 52)                0         
_________________________________________________________________
dense_12 (Dense)             (None, 10)                530       
=================================================================
Total params: 305,250
Trainable params: 305,250
Non-trainable params: 0
_________________________________________________________________
```

In [27]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

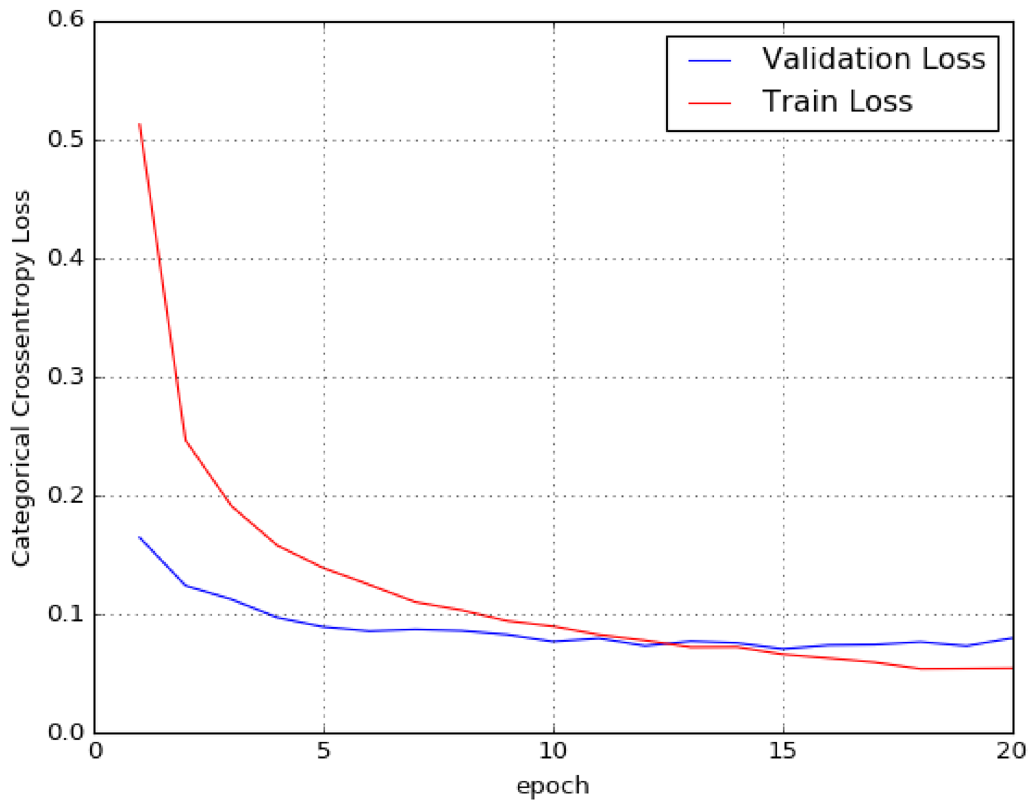
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.5129 -
acc: 0.8458 - val_loss: 0.1645 - val_acc: 0.9502
Epoch 2/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.2464 -
acc: 0.9262 - val_loss: 0.1239 - val_acc: 0.9634
Epoch 3/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.1909 -
acc: 0.9432 - val_loss: 0.1123 - val_acc: 0.9677
Epoch 4/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.1577 -
acc: 0.9533 - val_loss: 0.0969 - val_acc: 0.9705
Epoch 5/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.1387 -
acc: 0.9594 - val_loss: 0.0890 - val_acc: 0.9736
Epoch 6/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.1246 -
acc: 0.9627 - val_loss: 0.0855 - val_acc: 0.9765
Epoch 7/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1100 -
acc: 0.9670 - val_loss: 0.0869 - val_acc: 0.9757
Epoch 8/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.1032 -
acc: 0.9685 - val_loss: 0.0859 - val_acc: 0.9756
Epoch 9/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0941 -
acc: 0.9706 - val_loss: 0.0826 - val_acc: 0.9755
Epoch 10/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0895 -
acc: 0.9725 - val_loss: 0.0767 - val_acc: 0.9787
Epoch 11/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0824 -
acc: 0.9740 - val_loss: 0.0795 - val_acc: 0.9776
Epoch 12/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0777 -
acc: 0.9761 - val_loss: 0.0731 - val_acc: 0.9798
Epoch 13/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0720 -
acc: 0.9772 - val_loss: 0.0770 - val_acc: 0.9802
Epoch 14/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0722 -
acc: 0.9776 - val_loss: 0.0756 - val_acc: 0.9796
Epoch 15/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0659 -
acc: 0.9795 - val_loss: 0.0705 - val_acc: 0.9810
Epoch 16/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0627 -
acc: 0.9811 - val_loss: 0.0737 - val_acc: 0.9818
Epoch 17/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.0592
- acc: 0.9815 - val_loss: 0.0741 - val_acc: 0.9809
Epoch 18/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0537
- acc: 0.9834 - val_loss: 0.0764 - val_acc: 0.9798
```

```
Epoch 19/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.0539
- acc: 0.9831 - val_loss: 0.0730 - val_acc: 0.9814
Epoch 20/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0542 -
acc: 0.9835 - val_loss: 0.0797 - val_acc: 0.9802
```

In [28]:

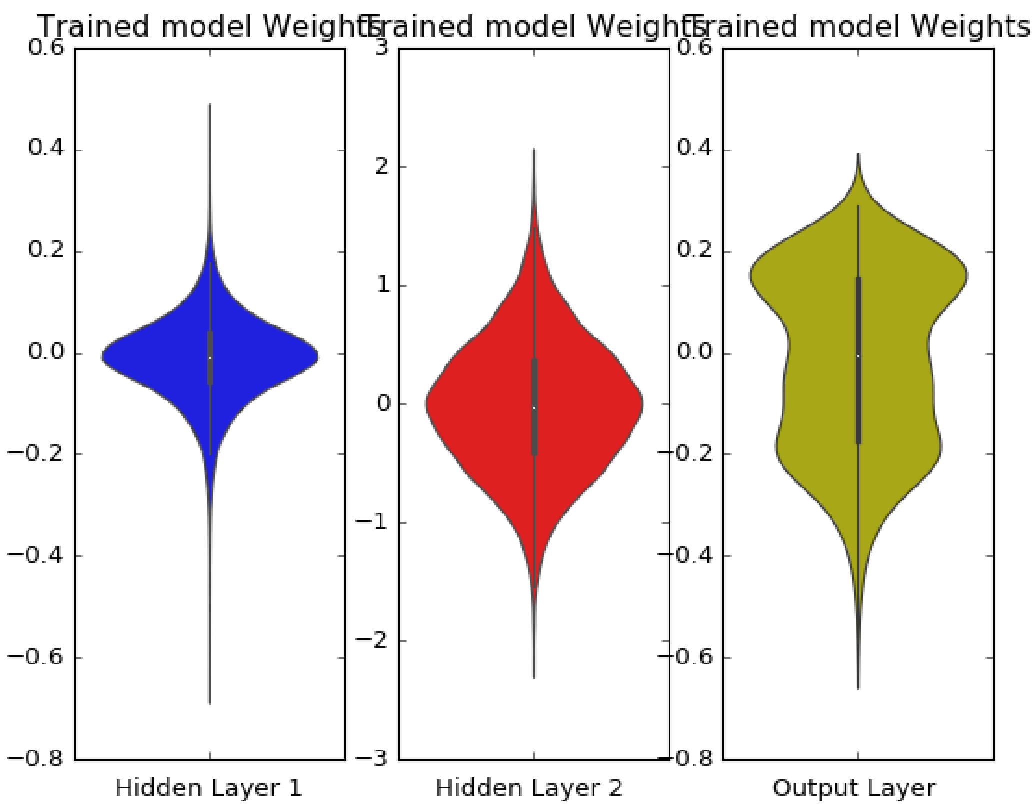
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.07972456046559565
Test accuracy: 0.9802
```



In [29]:

```
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()
```



In [ ]:

```

```

MLP + BatchNormalization + Dropout(rate = 0.5) + AdamOptimizer

with 2 hidden layers

In [30]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
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_13 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_3 (Batch (None, 364)               1456      
_________________________________________________________________
dropout_5 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_14 (Dense)             (None, 52)                18980     
_________________________________________________________________
batch_normalization_4 (Batch (None, 52)                208       
_________________________________________________________________
dropout_6 (Dropout)          (None, 52)                0         
_________________________________________________________________
dense_15 (Dense)             (None, 10)                530       
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

In [31]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

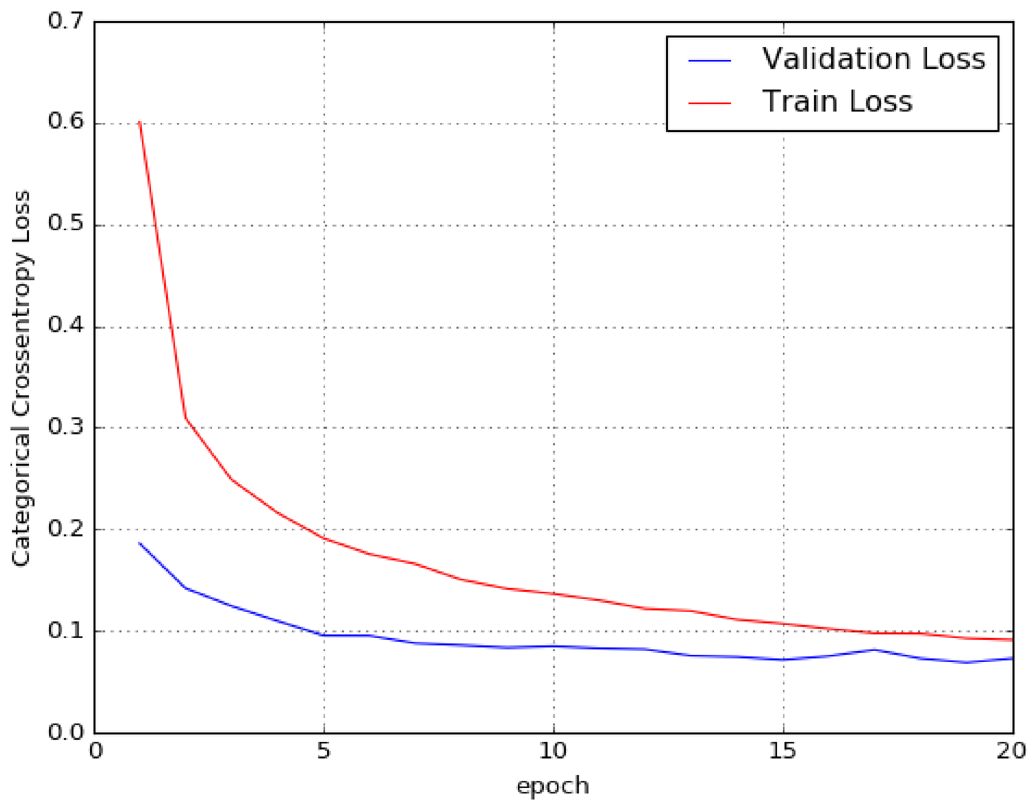
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.6007
- acc: 0.8222 - val_loss: 0.1861 - val_acc: 0.9449
Epoch 2/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.3093
- acc: 0.9088 - val_loss: 0.1419 - val_acc: 0.9576
Epoch 3/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.2488
- acc: 0.9279 - val_loss: 0.1245 - val_acc: 0.9627
Epoch 4/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.2162 -
acc: 0.9373 - val_loss: 0.1098 - val_acc: 0.9646
Epoch 5/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.1912 -
acc: 0.9440 - val_loss: 0.0955 - val_acc: 0.9707
Epoch 6/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.1756 -
acc: 0.9489 - val_loss: 0.0954 - val_acc: 0.9714
Epoch 7/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.1659
- acc: 0.9525 - val_loss: 0.0879 - val_acc: 0.9751
Epoch 8/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1504 -
acc: 0.9550 - val_loss: 0.0859 - val_acc: 0.9729
Epoch 9/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1415 -
acc: 0.9587 - val_loss: 0.0836 - val_acc: 0.9736
Epoch 10/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.1365 -
acc: 0.9601 - val_loss: 0.0850 - val_acc: 0.9747
Epoch 11/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.1301
- acc: 0.9619 - val_loss: 0.0830 - val_acc: 0.9765
Epoch 12/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.1219 -
acc: 0.9637 - val_loss: 0.0820 - val_acc: 0.9755
Epoch 13/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.1196
- acc: 0.9650 - val_loss: 0.0756 - val_acc: 0.9775
Epoch 14/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.1112
- acc: 0.9674 - val_loss: 0.0745 - val_acc: 0.9778
Epoch 15/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.1071
- acc: 0.9686 - val_loss: 0.0714 - val_acc: 0.9795
Epoch 16/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.1020
- acc: 0.9700 - val_loss: 0.0752 - val_acc: 0.9776
Epoch 17/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.0977
- acc: 0.9708 - val_loss: 0.0815 - val_acc: 0.9775
Epoch 18/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0974
- acc: 0.9700 - val_loss: 0.0728 - val_acc: 0.9782
```

```
Epoch 19/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0928 -
acc: 0.9723 - val_loss: 0.0688 - val_acc: 0.9795
Epoch 20/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0912 -
acc: 0.9728 - val_loss: 0.0729 - val_acc: 0.9806
```

In [33]:

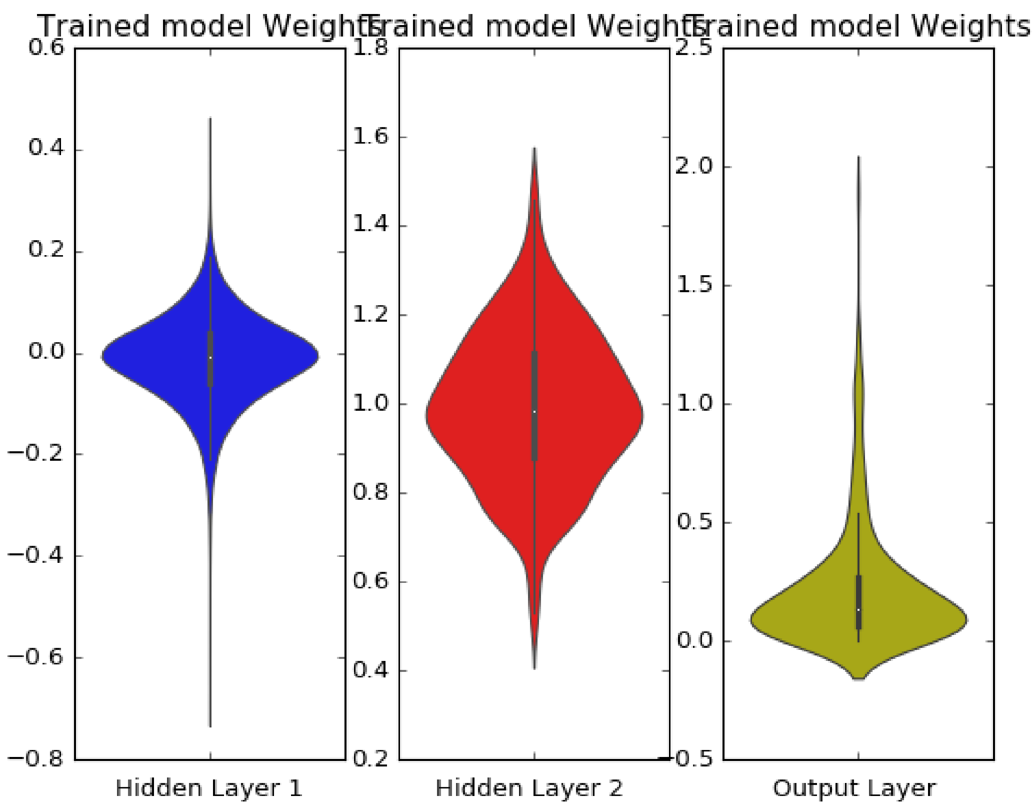
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.07286214664807195
Test accuracy: 0.9806
```



In [34]:

```
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()
```



In [ ]:

```

```

MLP + BatchNormalization + Dropout(rate = 0.3) + AdamOptimizer

with 2 hidden layers

In [36]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_16 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_5 (Batch (None, 364)               1456      
_________________________________________________________________
dropout_7 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_17 (Dense)             (None, 52)                18980     
_________________________________________________________________
batch_normalization_6 (Batch (None, 52)                208       
_________________________________________________________________
dropout_8 (Dropout)          (None, 52)                0         
_________________________________________________________________
dense_18 (Dense)             (None, 10)                530       
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

In [37]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

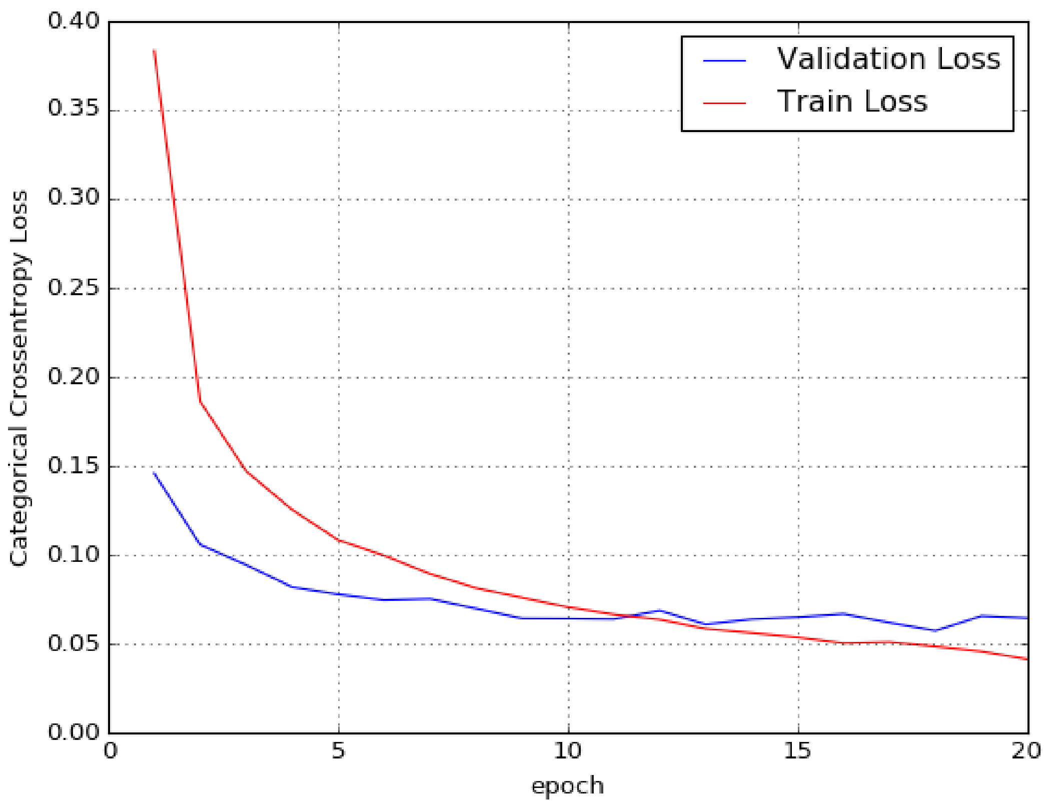
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.3831
- acc: 0.8881 - val_loss: 0.1457 - val_acc: 0.9558
Epoch 2/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.1861
- acc: 0.9443 - val_loss: 0.1056 - val_acc: 0.9669
Epoch 3/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.1469 -
acc: 0.9556 - val_loss: 0.0942 - val_acc: 0.9686
Epoch 4/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.1253 -
acc: 0.9615 - val_loss: 0.0818 - val_acc: 0.9751
Epoch 5/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.1083 -
acc: 0.9672 - val_loss: 0.0777 - val_acc: 0.9756
Epoch 6/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0995 -
acc: 0.9694 - val_loss: 0.0744 - val_acc: 0.9778
Epoch 7/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0892 -
acc: 0.9716 - val_loss: 0.0752 - val_acc: 0.9778
Epoch 8/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0813 -
acc: 0.9748 - val_loss: 0.0697 - val_acc: 0.9776
Epoch 9/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0758 -
acc: 0.9757 - val_loss: 0.0642 - val_acc: 0.9811
Epoch 10/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0705 -
acc: 0.9784 - val_loss: 0.0641 - val_acc: 0.9812
Epoch 11/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0664 -
acc: 0.9789 - val_loss: 0.0638 - val_acc: 0.9805
Epoch 12/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0636
- acc: 0.9799 - val_loss: 0.0686 - val_acc: 0.9791
Epoch 13/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.0583 -
acc: 0.9817 - val_loss: 0.0608 - val_acc: 0.9807
Epoch 14/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0560 -
acc: 0.9823 - val_loss: 0.0638 - val_acc: 0.9823
Epoch 15/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0535 -
acc: 0.9825 - val_loss: 0.0649 - val_acc: 0.9817
Epoch 16/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0503 -
acc: 0.9842 - val_loss: 0.0667 - val_acc: 0.9808
Epoch 17/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.0509
- acc: 0.9833 - val_loss: 0.0618 - val_acc: 0.9816
Epoch 18/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0484 -
acc: 0.9847 - val_loss: 0.0572 - val_acc: 0.9831
```

```
Epoch 19/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0456 -
acc: 0.9844 - val_loss: 0.0655 - val_acc: 0.9818
Epoch 20/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0412 -
acc: 0.9859 - val_loss: 0.0644 - val_acc: 0.9825
```

In [38]:

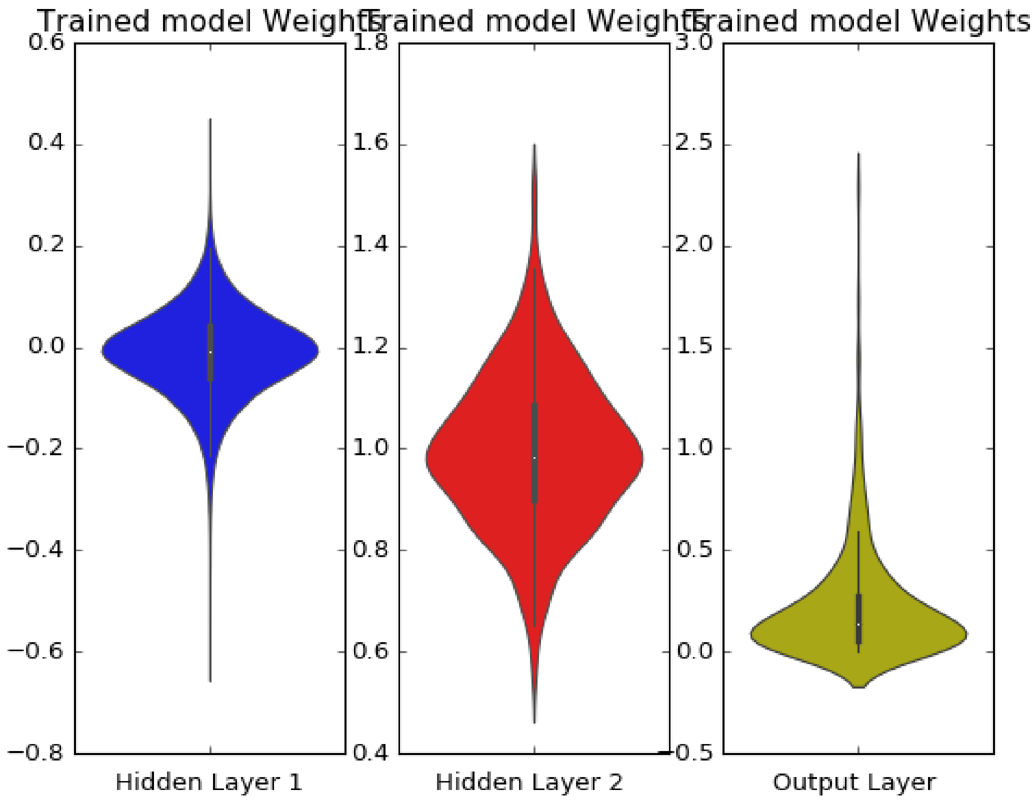
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.06437753892600304
Test accuracy: 0.9825
```



In [39]:

```
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()
```



MLP + ReLU + ADAM with 3 layers without Dropout and Batch Normalisation

In [40]:

```
model_relu = Sequential()
model_relu.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_relu.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

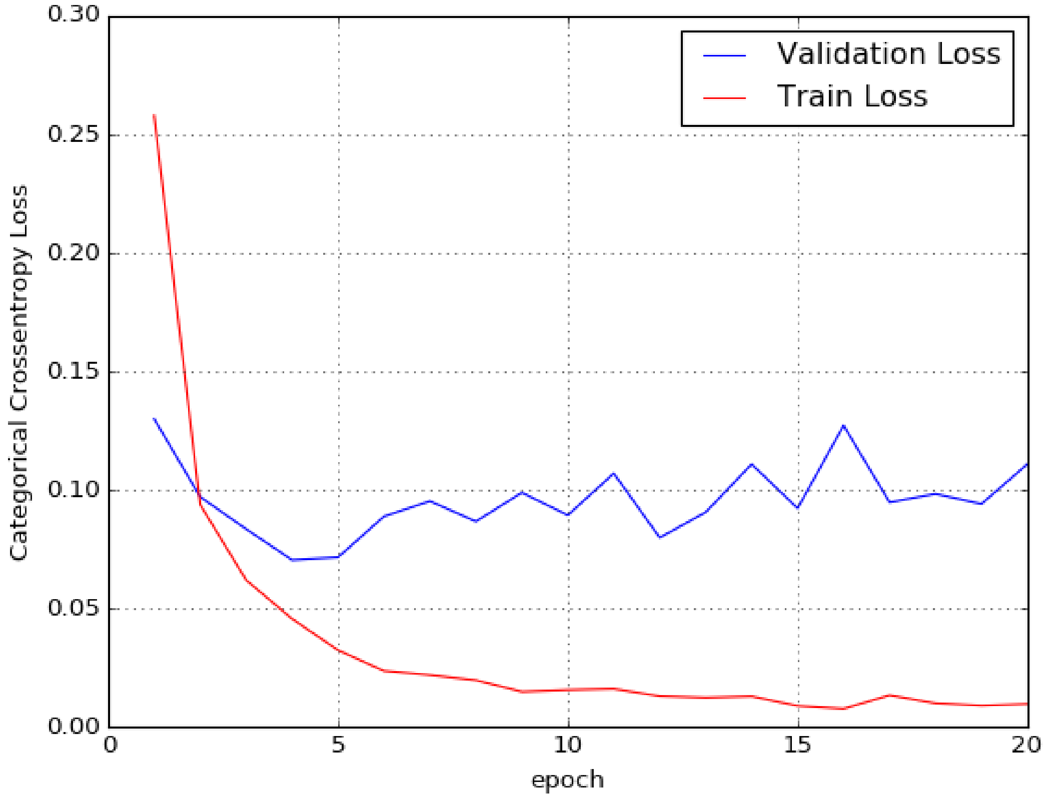
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_19 (Dense)             (None, 364)               285740
_________________________________________________________________
dense_20 (Dense)             (None, 128)               46720
_________________________________________________________________
dense_21 (Dense)             (None, 52)                6708
_________________________________________________________________
dense_22 (Dense)             (None, 10)                530
=================================================================
Total params: 339,698
Trainable params: 339,698
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2579 -
acc: 0.9240 - val_loss: 0.1299 - val_acc: 0.9621
Epoch 2/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0937 -
acc: 0.9717 - val_loss: 0.0970 - val_acc: 0.9708
Epoch 3/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0619 -
acc: 0.9809 - val_loss: 0.0834 - val_acc: 0.9741
Epoch 4/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0455 -
acc: 0.9858 - val_loss: 0.0703 - val_acc: 0.9798
Epoch 5/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0323 -
acc: 0.9893 - val_loss: 0.0714 - val_acc: 0.9778
Epoch 6/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0235 -
acc: 0.9924 - val_loss: 0.0888 - val_acc: 0.9745
Epoch 7/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0218 -
acc: 0.9928 - val_loss: 0.0952 - val_acc: 0.9740
Epoch 8/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0196 -
acc: 0.9939 - val_loss: 0.0866 - val_acc: 0.9768
Epoch 9/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0147 -
acc: 0.9951 - val_loss: 0.0988 - val_acc: 0.9772
Epoch 10/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0155 -
```

```
acc: 0.9947 - val_loss: 0.0893 - val_acc: 0.9792
Epoch 11/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.0159
- acc: 0.9945 - val_loss: 0.1069 - val_acc: 0.9745
Epoch 12/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0129 -
acc: 0.9958 - val_loss: 0.0797 - val_acc: 0.9806
Epoch 13/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0122 -
acc: 0.9958 - val_loss: 0.0905 - val_acc: 0.9810
Epoch 14/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0128 -
acc: 0.9958 - val_loss: 0.1108 - val_acc: 0.9765
Epoch 15/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0087 -
acc: 0.9971 - val_loss: 0.0922 - val_acc: 0.9805
Epoch 16/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0075 -
acc: 0.9977 - val_loss: 0.1271 - val_acc: 0.9749
Epoch 17/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0132 -
acc: 0.9959 - val_loss: 0.0947 - val_acc: 0.9799
Epoch 18/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0098 -
acc: 0.9967 - val_loss: 0.0982 - val_acc: 0.9793
Epoch 19/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0089 -
acc: 0.9970 - val_loss: 0.0940 - val_acc: 0.9806
Epoch 20/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0095 -
acc: 0.9970 - val_loss: 0.1108 - val_acc: 0.9788
```

In [41]:

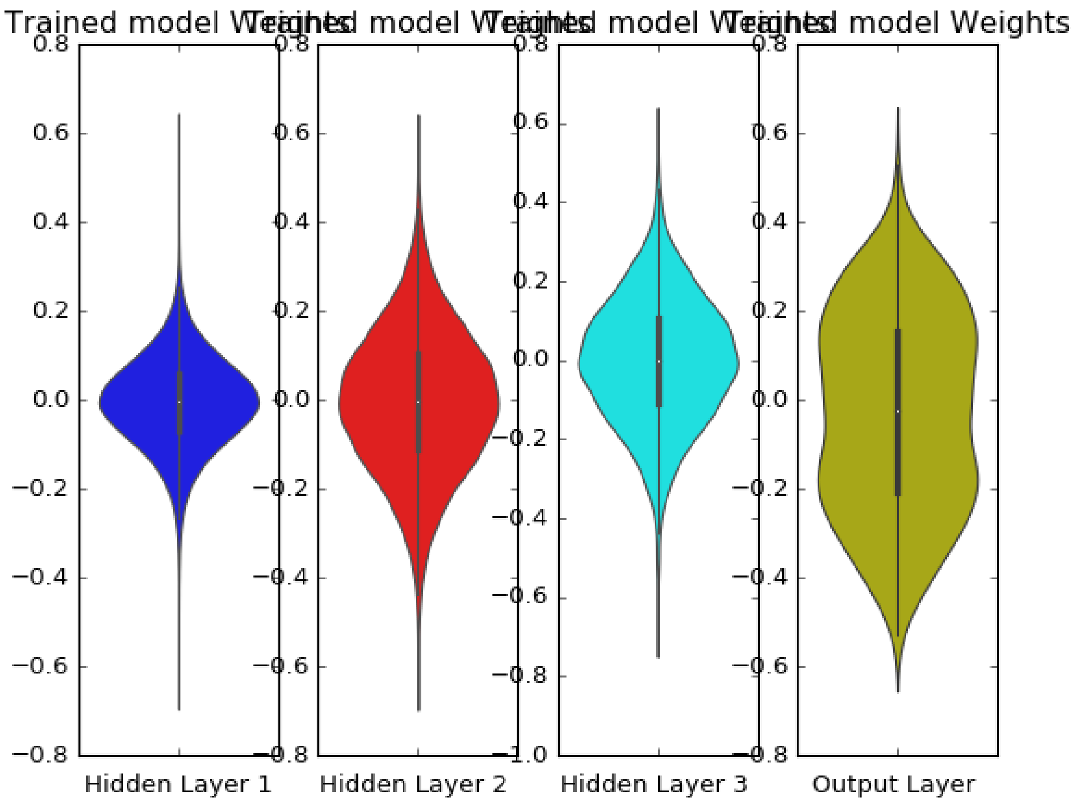
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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


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

```
Test score: 0.1108168175617167
Test accuracy: 0.9788
```



In [43]:

```
w_after = model_relu.get_weights()

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


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

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

plt.subplot(1, 4, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='cyan')
plt.xlabel('Hidden Layer 3 ')


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



MLP + Batch-Norm on 3 hidden Layers + AdamOptimizer

In [44]:

```
# 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=√
# 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(364, activation='relu', input_shape=(input_dim,), kernel_initializer=
model_batch.add(BatchNormalization())

model_batch.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
model_batch.add(BatchNormalization())

model_batch.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 364)               285740
_________________________________________________________________
batch_normalization_7 (Batch (None, 364)               1456
_________________________________________________________________
dense_24 (Dense)             (None, 128)               46720
_________________________________________________________________
batch_normalization_8 (Batch (None, 128)               512
_________________________________________________________________
dense_25 (Dense)             (None, 52)                6708
_________________________________________________________________
batch_normalization_9 (Batch (None, 52)                208
_________________________________________________________________
dense_26 (Dense)             (None, 10)                530
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

In [45]:

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

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose
```

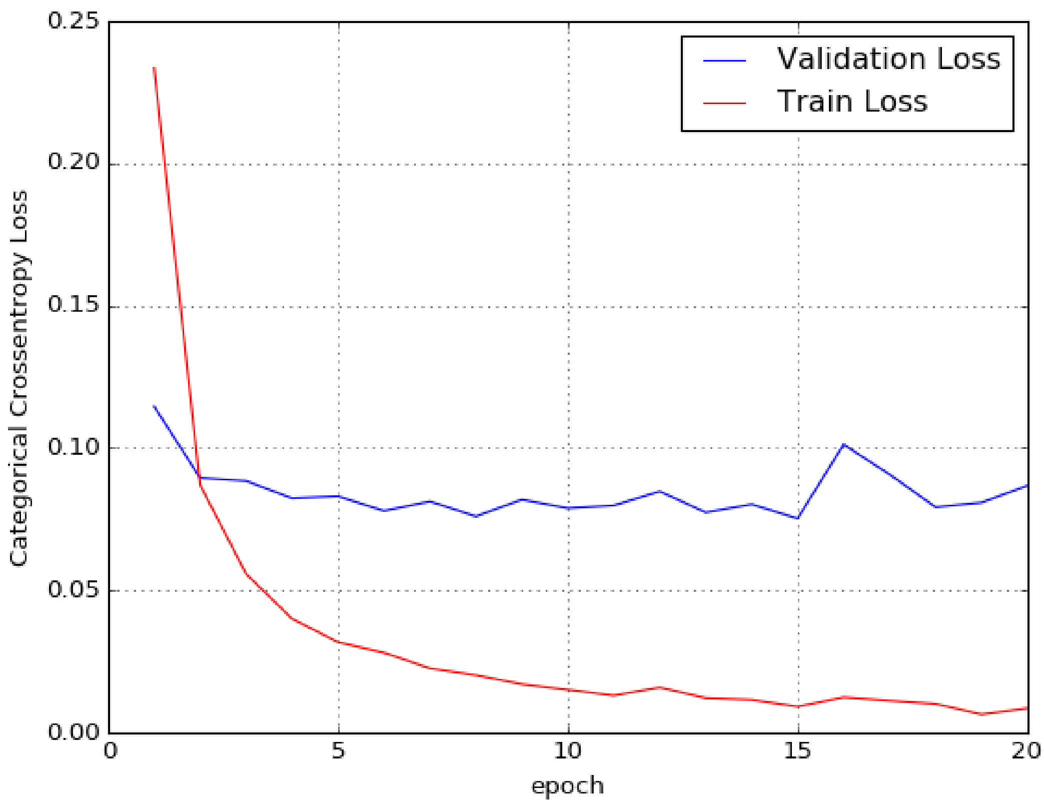
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.2334
- acc: 0.9338 - val_loss: 0.1146 - val_acc: 0.9651
Epoch 2/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0870
- acc: 0.9744 - val_loss: 0.0894 - val_acc: 0.9718
Epoch 3/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0557
- acc: 0.9826 - val_loss: 0.0884 - val_acc: 0.9715
Epoch 4/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0400 -
acc: 0.9876 - val_loss: 0.0823 - val_acc: 0.9760
Epoch 5/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0318 -
acc: 0.9901 - val_loss: 0.0830 - val_acc: 0.9758
Epoch 6/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.0281
- acc: 0.9907 - val_loss: 0.0779 - val_acc: 0.9770
Epoch 7/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0226
- acc: 0.9930 - val_loss: 0.0812 - val_acc: 0.9746
Epoch 8/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.0202
- acc: 0.9934 - val_loss: 0.0760 - val_acc: 0.9788
Epoch 9/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.0170
- acc: 0.9944 - val_loss: 0.0818 - val_acc: 0.9777
Epoch 10/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0150
- acc: 0.9949 - val_loss: 0.0788 - val_acc: 0.9778
Epoch 11/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.0130
- acc: 0.9959 - val_loss: 0.0798 - val_acc: 0.9788
Epoch 12/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0159
- acc: 0.9948 - val_loss: 0.0847 - val_acc: 0.9777
Epoch 13/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0121 -
acc: 0.9961 - val_loss: 0.0773 - val_acc: 0.9794
Epoch 14/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0115 -
acc: 0.9960 - val_loss: 0.0802 - val_acc: 0.9798
Epoch 15/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0091 -
acc: 0.9970 - val_loss: 0.0752 - val_acc: 0.9803
Epoch 16/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0124 -
acc: 0.9956 - val_loss: 0.1012 - val_acc: 0.9752
Epoch 17/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0112 -
acc: 0.9961 - val_loss: 0.0907 - val_acc: 0.9775
Epoch 18/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.0101
- acc: 0.9966 - val_loss: 0.0792 - val_acc: 0.9799
```

```
Epoch 19/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0064
- acc: 0.9980 - val_loss: 0.0808 - val_acc: 0.9787
Epoch 20/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0085
- acc: 0.9972 - val_loss: 0.0867 - val_acc: 0.9795
```

In [46]:

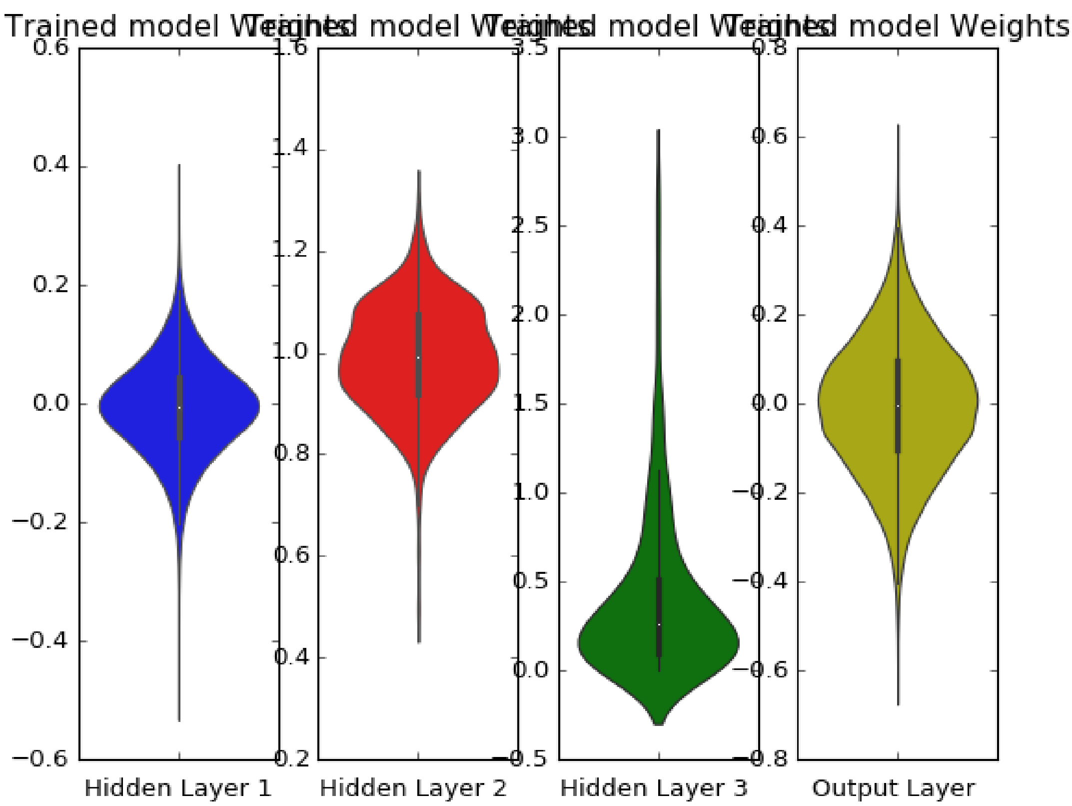
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.08674843371730385
Test accuracy: 0.9795
```



In [47]:

```
w_after = model_batch.get_weights()

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

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

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

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

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



MLP + Dropout(rate = 0.5) + AdamOptimizer with 3 hidden layers

In [48]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_drop.add(Dropout(0.5))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_27 (Dense)             (None, 364)               285740    
_________________________________________________________________
dropout_9 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_28 (Dense)             (None, 128)               46720     
_________________________________________________________________
dropout_10 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_29 (Dense)             (None, 52)                6708      
_________________________________________________________________
dropout_11 (Dropout)         (None, 52)                0         
_________________________________________________________________
dense_30 (Dense)             (None, 10)                530       
=================================================================
Total params: 339,698
Trainable params: 339,698
Non-trainable params: 0
_________________________________________________________________
```

In [49]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

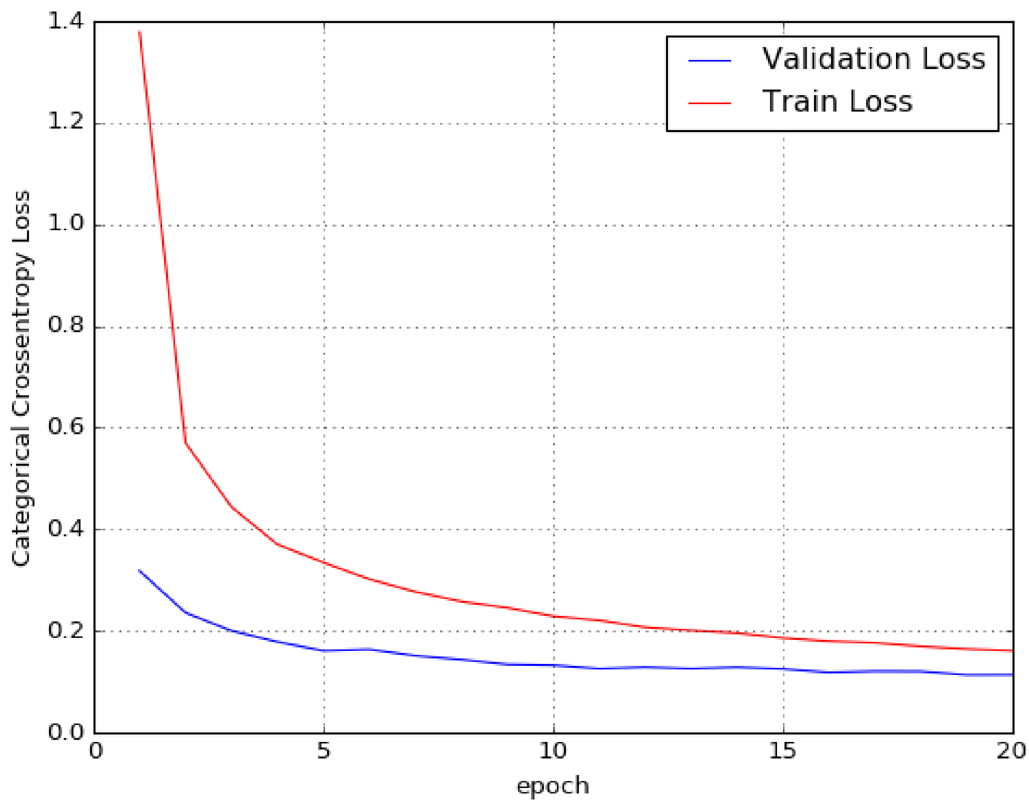
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 100us/step - loss: 1.3778
- acc: 0.5752 - val_loss: 0.3179 - val_acc: 0.9180
Epoch 2/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.5697 -
acc: 0.8389 - val_loss: 0.2357 - val_acc: 0.9406
Epoch 3/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.4437
- acc: 0.8838 - val_loss: 0.2002 - val_acc: 0.9455
Epoch 4/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.3701
- acc: 0.9034 - val_loss: 0.1784 - val_acc: 0.9543
Epoch 5/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.3346
- acc: 0.9151 - val_loss: 0.1610 - val_acc: 0.9578
Epoch 6/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.3017 -
acc: 0.9235 - val_loss: 0.1637 - val_acc: 0.9576
Epoch 7/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.2769 -
acc: 0.9303 - val_loss: 0.1512 - val_acc: 0.9607
Epoch 8/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.2577
- acc: 0.9356 - val_loss: 0.1434 - val_acc: 0.9621
Epoch 9/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.2455
- acc: 0.9385 - val_loss: 0.1340 - val_acc: 0.9660
Epoch 10/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.2290 -
acc: 0.9429 - val_loss: 0.1325 - val_acc: 0.9661
Epoch 11/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.2208
- acc: 0.9442 - val_loss: 0.1257 - val_acc: 0.9683
Epoch 12/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.2071
- acc: 0.9477 - val_loss: 0.1286 - val_acc: 0.9702
Epoch 13/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.2008
- acc: 0.9485 - val_loss: 0.1258 - val_acc: 0.9699
Epoch 14/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.1956 -
acc: 0.9515 - val_loss: 0.1284 - val_acc: 0.9693
Epoch 15/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.1862
- acc: 0.9537 - val_loss: 0.1253 - val_acc: 0.9703
Epoch 16/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.1797
- acc: 0.9547 - val_loss: 0.1183 - val_acc: 0.9694
Epoch 17/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.1766
- acc: 0.9546 - val_loss: 0.1206 - val_acc: 0.9718
Epoch 18/20
60000/60000 [==============================] - 7s 117us/step - loss: 0.1697
- acc: 0.9570 - val_loss: 0.1203 - val_acc: 0.9703
```

```
Epoch 19/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.1644
- acc: 0.9590 - val_loss: 0.1134 - val_acc: 0.9734
Epoch 20/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.1611 -
acc: 0.9606 - val_loss: 0.1137 - val_acc: 0.9741
```

In [50]:

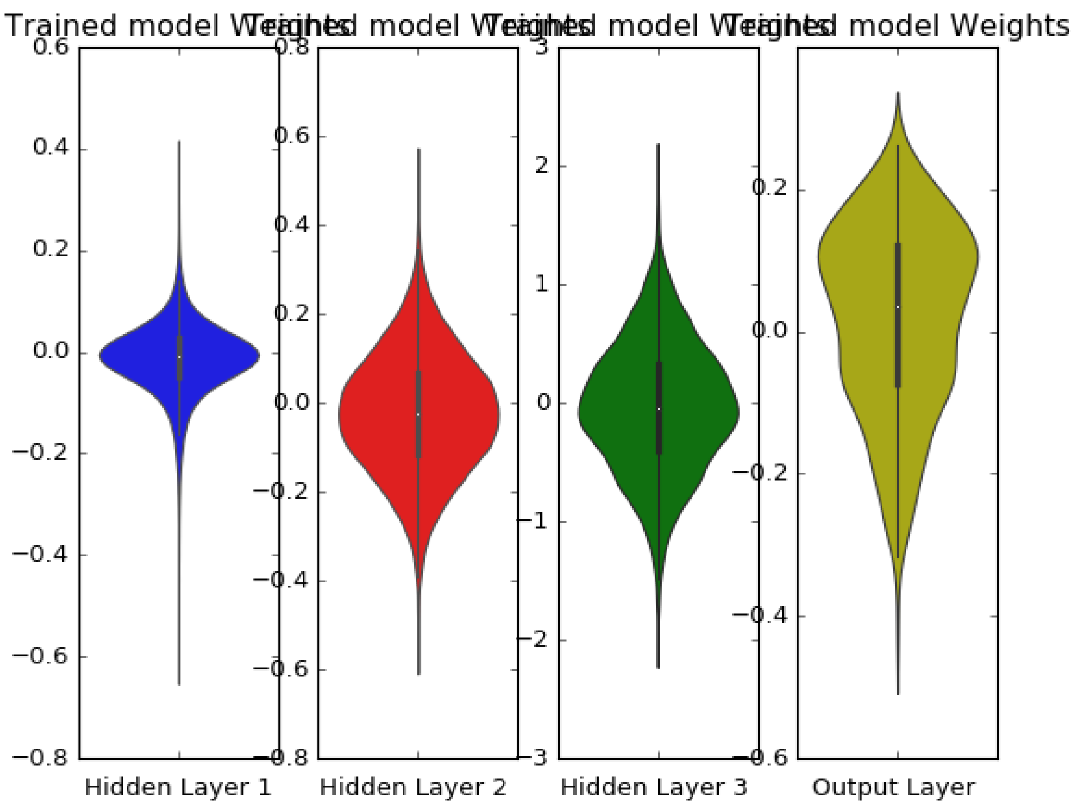
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.11365028675964568
Test accuracy: 0.9741
```



In [51]:

```
w_after = model_drop.get_weights()

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


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

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

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

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



MLP + Dropout(rate = 0.3) + AdamOptimizer with 3 hidden layers

In [52]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(Dropout(0.3))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_drop.add(Dropout(0.3))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.3))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_31 (Dense)             (None, 364)               285740    
_________________________________________________________________
dropout_12 (Dropout)         (None, 364)               0         
_________________________________________________________________
dense_32 (Dense)             (None, 128)               46720     
_________________________________________________________________
dropout_13 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_33 (Dense)             (None, 52)                6708      
_________________________________________________________________
dropout_14 (Dropout)         (None, 52)                0         
_________________________________________________________________
dense_34 (Dense)             (None, 10)                530       
=================================================================
Total params: 339,698
Trainable params: 339,698
Non-trainable params: 0
_________________________________________________________________
```

In [53]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

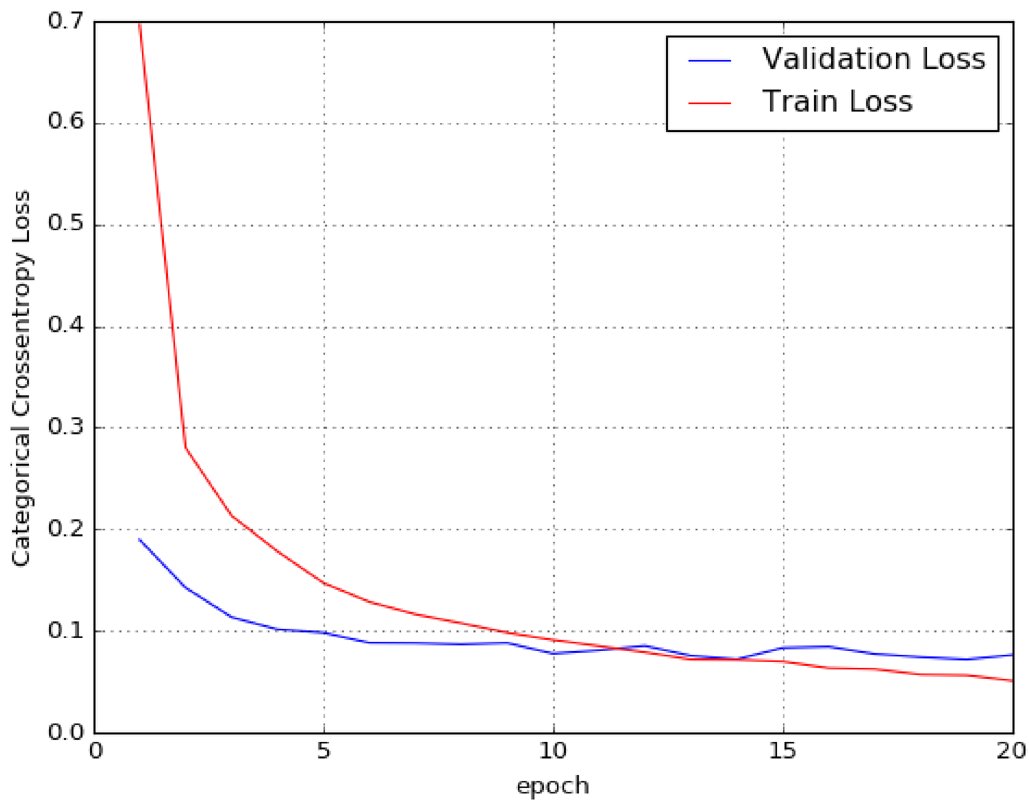
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.6964
- acc: 0.7865 - val_loss: 0.1897 - val_acc: 0.9470
Epoch 2/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.2802
- acc: 0.9218 - val_loss: 0.1425 - val_acc: 0.9601
Epoch 3/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.2134 -
acc: 0.9400 - val_loss: 0.1135 - val_acc: 0.9674
Epoch 4/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.1782 -
acc: 0.9506 - val_loss: 0.1015 - val_acc: 0.9721
Epoch 5/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.1472 -
acc: 0.9601 - val_loss: 0.0982 - val_acc: 0.9723
Epoch 6/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.1285 -
acc: 0.9646 - val_loss: 0.0883 - val_acc: 0.9757
Epoch 7/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.1164 -
acc: 0.9681 - val_loss: 0.0880 - val_acc: 0.9763
Epoch 8/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1076 -
acc: 0.9702 - val_loss: 0.0868 - val_acc: 0.9761
Epoch 9/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0982 -
acc: 0.9726 - val_loss: 0.0881 - val_acc: 0.9772
Epoch 10/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.0911
- acc: 0.9751 - val_loss: 0.0777 - val_acc: 0.9785
Epoch 11/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.0851 -
acc: 0.9759 - val_loss: 0.0810 - val_acc: 0.9799
Epoch 12/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0790
- acc: 0.9782 - val_loss: 0.0853 - val_acc: 0.9786
Epoch 13/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.0719 -
acc: 0.9791 - val_loss: 0.0756 - val_acc: 0.9799
Epoch 14/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0716 -
acc: 0.9803 - val_loss: 0.0721 - val_acc: 0.9794
Epoch 15/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0699
- acc: 0.9804 - val_loss: 0.0834 - val_acc: 0.9789
Epoch 16/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.0635 -
acc: 0.9818 - val_loss: 0.0845 - val_acc: 0.9799
Epoch 17/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0625 -
acc: 0.9828 - val_loss: 0.0773 - val_acc: 0.9803
Epoch 18/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.0570
- acc: 0.9835 - val_loss: 0.0743 - val_acc: 0.9823
```

```
Epoch 19/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.0564 -
acc: 0.9842 - val_loss: 0.0719 - val_acc: 0.9811
Epoch 20/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0509 -
acc: 0.9853 - val_loss: 0.0763 - val_acc: 0.9806
```

In [55]:

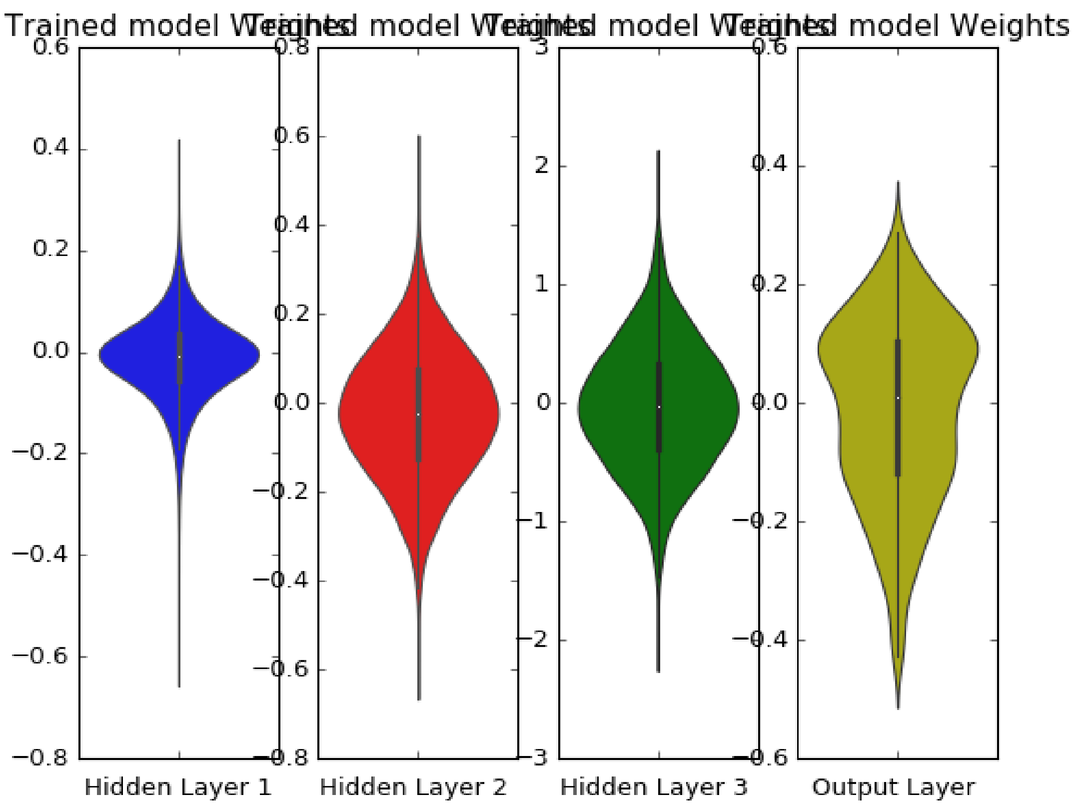
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.07634433440160543
Test accuracy: 0.9806
```



In [56]:

```
w_after = model_drop.get_weights()

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


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

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

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

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



MLP + BatchNormalization + Dropout(rate = 0.5) + AdamOptimizer with 3 hidden layers

In [57]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))


model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
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_35 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_10 (Batc (None, 364)               1456      
_________________________________________________________________
dropout_15 (Dropout)         (None, 364)               0         
_________________________________________________________________
dense_36 (Dense)             (None, 128)               46720     
_________________________________________________________________
batch_normalization_11 (Batc (None, 128)               512       
_________________________________________________________________
dropout_16 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_37 (Dense)             (None, 52)                6708      
_________________________________________________________________
batch_normalization_12 (Batc (None, 52)                208       
_________________________________________________________________
dropout_17 (Dropout)         (None, 52)                0         
_________________________________________________________________
dense_38 (Dense)             (None, 10)                530       
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

In [58]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

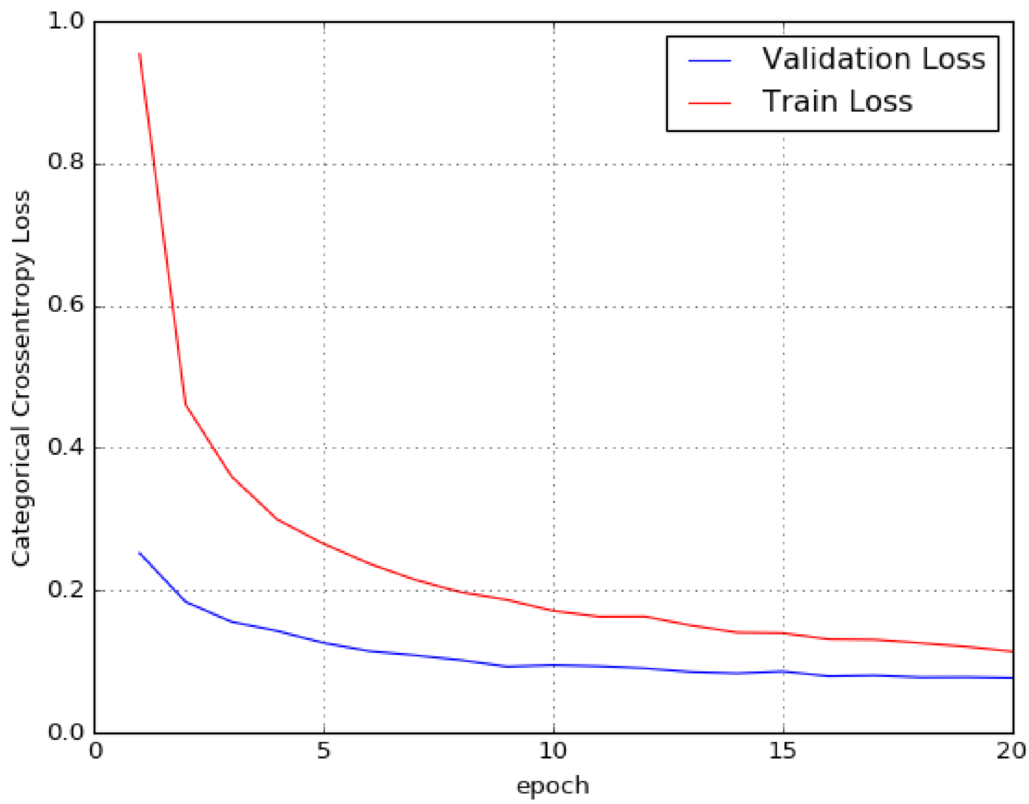
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.9536
- acc: 0.7001 - val_loss: 0.2521 - val_acc: 0.9256
Epoch 2/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.4609
- acc: 0.8654 - val_loss: 0.1837 - val_acc: 0.9456
Epoch 3/20
60000/60000 [==============================] - 7s 108us/step - loss: 0.3600
- acc: 0.8969 - val_loss: 0.1557 - val_acc: 0.9529
Epoch 4/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.2992
- acc: 0.9157 - val_loss: 0.1426 - val_acc: 0.9560
Epoch 5/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.2653
- acc: 0.9260 - val_loss: 0.1260 - val_acc: 0.9624
Epoch 6/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.2374
- acc: 0.9345 - val_loss: 0.1145 - val_acc: 0.9660
Epoch 7/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.2146
- acc: 0.9404 - val_loss: 0.1085 - val_acc: 0.9691
Epoch 8/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.1971
- acc: 0.9455 - val_loss: 0.1016 - val_acc: 0.9689
Epoch 9/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.1864
- acc: 0.9488 - val_loss: 0.0927 - val_acc: 0.9721
Epoch 10/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.1710
- acc: 0.9527 - val_loss: 0.0948 - val_acc: 0.9714
Epoch 11/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.1629
- acc: 0.9552 - val_loss: 0.0933 - val_acc: 0.9722
Epoch 12/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.1631
- acc: 0.9558 - val_loss: 0.0903 - val_acc: 0.9742
Epoch 13/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.1505
- acc: 0.9584 - val_loss: 0.0852 - val_acc: 0.9749
Epoch 14/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.1406
- acc: 0.9608 - val_loss: 0.0831 - val_acc: 0.9766
Epoch 15/20
60000/60000 [==============================] - 8s 130us/step - loss: 0.1397
- acc: 0.9609 - val_loss: 0.0859 - val_acc: 0.9755
Epoch 16/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.1312
- acc: 0.9641 - val_loss: 0.0792 - val_acc: 0.9766
Epoch 17/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.1305
- acc: 0.9632 - val_loss: 0.0803 - val_acc: 0.9778
Epoch 18/20
60000/60000 [==============================] - 8s 128us/step - loss: 0.1257
- acc: 0.9645 - val_loss: 0.0776 - val_acc: 0.9787
```

```
Epoch 19/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.1207
- acc: 0.9661 - val_loss: 0.0778 - val_acc: 0.9782
Epoch 20/20
60000/60000 [==============================] - 9s 148us/step - loss: 0.1140
- acc: 0.9693 - val_loss: 0.0768 - val_acc: 0.9797
```

In [59]:

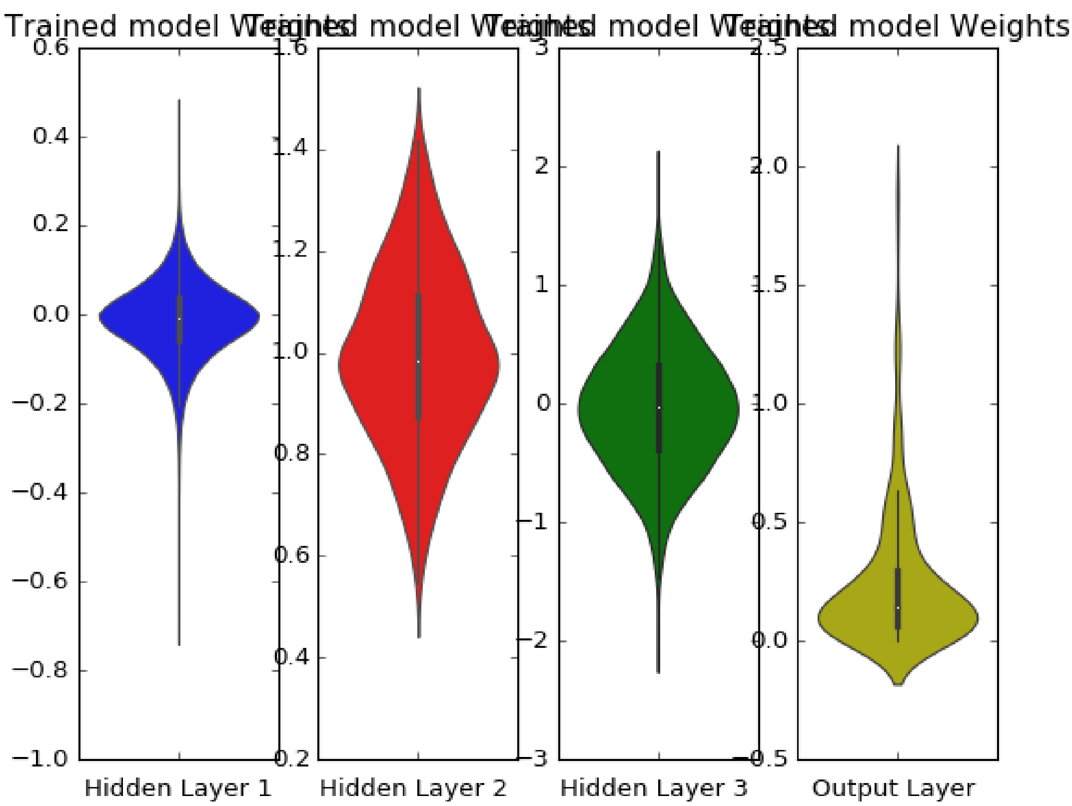
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.07683449132365058
Test accuracy: 0.9797
```



In [60]:

```
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, 4, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

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

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



MLP + BatchNormalization + Dropout(rate = 0.3) + AdamOptimizer with 3 hidden layers

In [61]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))


model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_39 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_13 (Batc (None, 364)               1456      
_________________________________________________________________
dropout_18 (Dropout)         (None, 364)               0         
_________________________________________________________________
dense_40 (Dense)             (None, 128)               46720     
_________________________________________________________________
batch_normalization_14 (Batc (None, 128)               512       
_________________________________________________________________
dropout_19 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_41 (Dense)             (None, 52)                6708      
_________________________________________________________________
batch_normalization_15 (Batc (None, 52)                208       
_________________________________________________________________
dropout_20 (Dropout)         (None, 52)                0         
_________________________________________________________________
dense_42 (Dense)             (None, 10)                530       
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

In [62]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

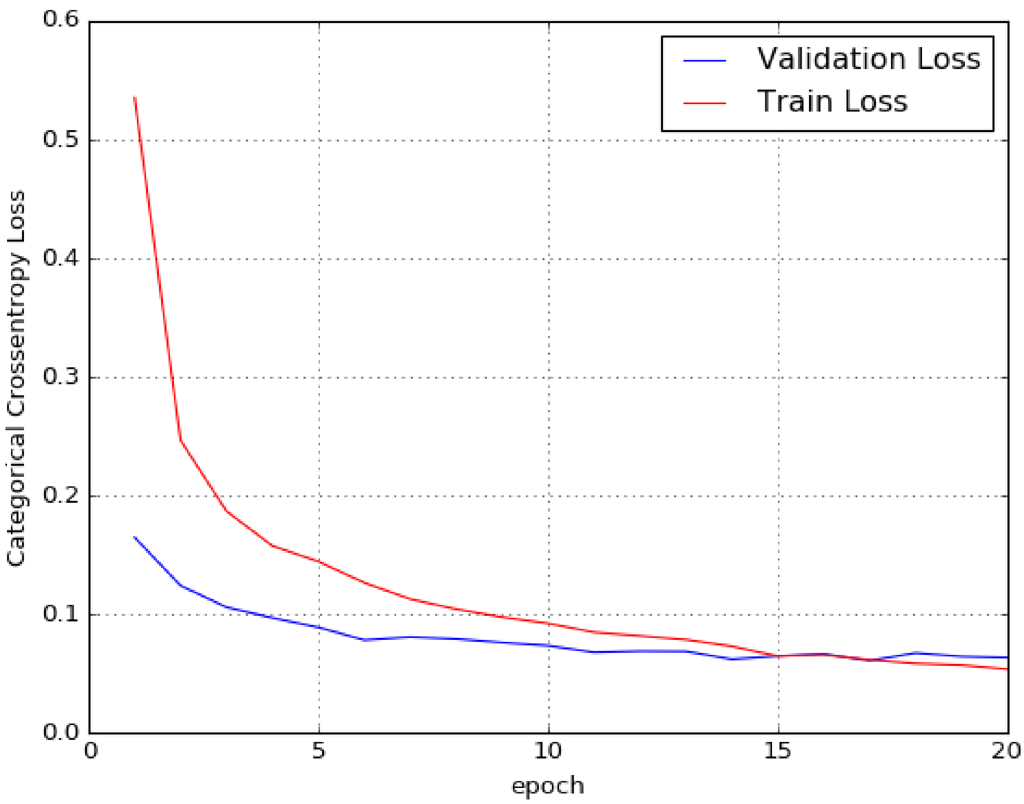
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.5354
- acc: 0.8428 - val_loss: 0.1646 - val_acc: 0.9491
Epoch 2/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.2464
- acc: 0.9281 - val_loss: 0.1239 - val_acc: 0.9613
Epoch 3/20
60000/60000 [==============================] - 9s 155us/step - loss: 0.1866
- acc: 0.9460 - val_loss: 0.1056 - val_acc: 0.9687
Epoch 4/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.1574
- acc: 0.9528 - val_loss: 0.0967 - val_acc: 0.9694
Epoch 5/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.1442
- acc: 0.9573 - val_loss: 0.0887 - val_acc: 0.9729
Epoch 6/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.1263
- acc: 0.9619 - val_loss: 0.0780 - val_acc: 0.9776
Epoch 7/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.1125
- acc: 0.9672 - val_loss: 0.0806 - val_acc: 0.9757
Epoch 8/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.1040
- acc: 0.9682 - val_loss: 0.0790 - val_acc: 0.9764
Epoch 9/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0973
- acc: 0.9709 - val_loss: 0.0759 - val_acc: 0.9773
Epoch 10/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.0921
- acc: 0.9727 - val_loss: 0.0733 - val_acc: 0.9782
Epoch 11/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.0845
- acc: 0.9739 - val_loss: 0.0677 - val_acc: 0.9794
Epoch 12/20
60000/60000 [==============================] - 10s 162us/step - loss: 0.0815
- acc: 0.9748 - val_loss: 0.0687 - val_acc: 0.9799
Epoch 13/20
60000/60000 [==============================] - 9s 148us/step - loss: 0.0783
- acc: 0.9760 - val_loss: 0.0685 - val_acc: 0.9801
Epoch 14/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.0727
- acc: 0.9775 - val_loss: 0.0618 - val_acc: 0.9808
Epoch 15/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.0645
- acc: 0.9798 - val_loss: 0.0645 - val_acc: 0.9813
Epoch 16/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.0656
- acc: 0.9798 - val_loss: 0.0663 - val_acc: 0.9823
Epoch 17/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.0613
- acc: 0.9808 - val_loss: 0.0606 - val_acc: 0.9823
Epoch 18/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.0582
- acc: 0.9815 - val_loss: 0.0671 - val_acc: 0.9814
```

```
Epoch 19/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0568
- acc: 0.9829 - val_loss: 0.0642 - val_acc: 0.9817
Epoch 20/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.0535
- acc: 0.9835 - val_loss: 0.0632 - val_acc: 0.9835
```

In [63]:

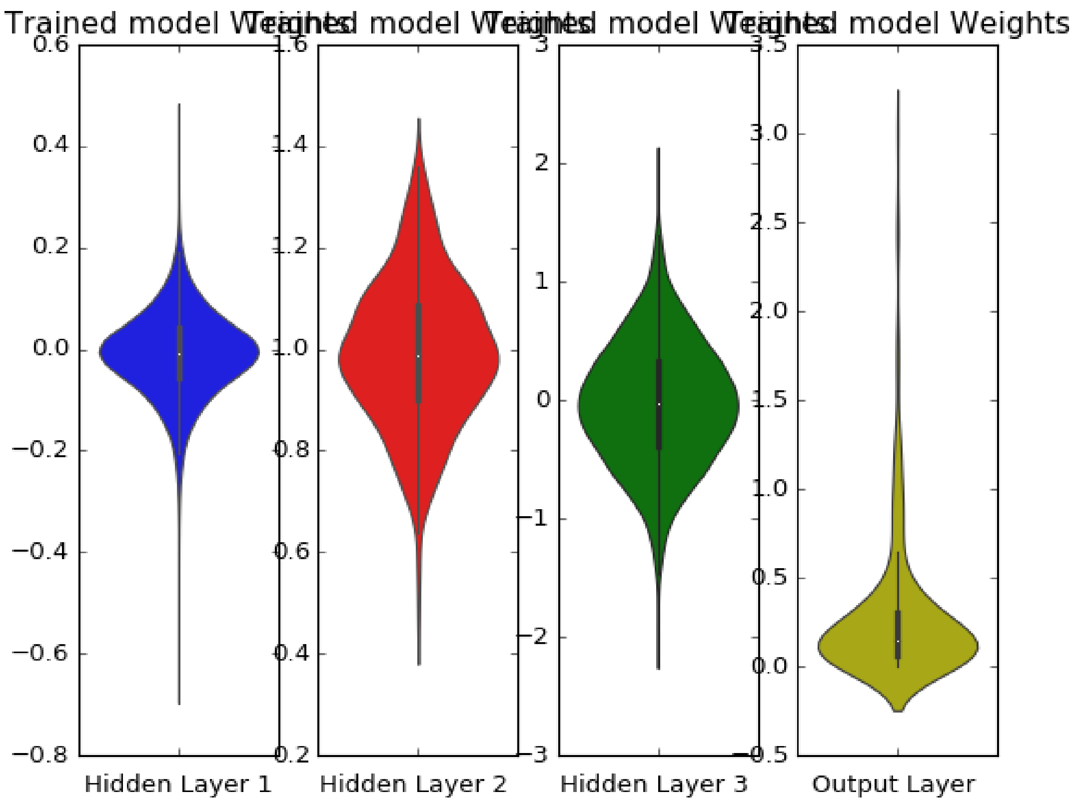
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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

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

```
Test score: 0.06322702438611887
Test accuracy: 0.9835
```



In [64]:

```
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, 4, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

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

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



MLP + ReLU + ADAM with 5 layers without Dropout and Batch Normalisation

In [65]:

```
model_relu = Sequential()
model_relu.add(Dense(256, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_relu.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

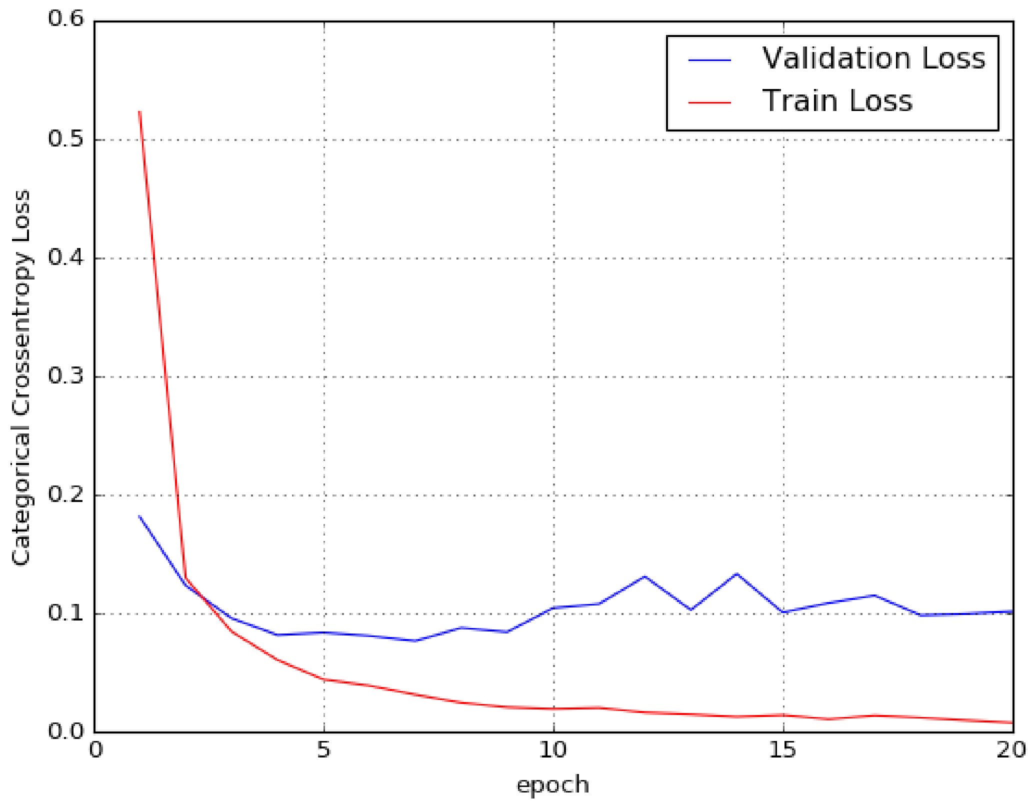
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_43 (Dense)             (None, 256)               200960
_________________________________________________________________
dense_44 (Dense)             (None, 128)               32896
_________________________________________________________________
dense_45 (Dense)             (None, 64)                8256
_________________________________________________________________
dense_46 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_47 (Dense)             (None, 16)                528
_________________________________________________________________
dense_48 (Dense)             (None, 10)                170
=================================================================
Total params: 244,890
Trainable params: 244,890
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.5227 -
acc: 0.8537 - val_loss: 0.1814 - val_acc: 0.9456
Epoch 2/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1297 -
acc: 0.9628 - val_loss: 0.1234 - val_acc: 0.9640
Epoch 3/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0845 -
acc: 0.9744 - val_loss: 0.0956 - val_acc: 0.9720
Epoch 4/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0605 -
acc: 0.9817 - val_loss: 0.0815 - val_acc: 0.9738
Epoch 5/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0440 -
acc: 0.9865 - val_loss: 0.0836 - val_acc: 0.9751
Epoch 6/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.0388 -
acc: 0.9875 - val_loss: 0.0808 - val_acc: 0.9774
Epoch 7/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0312 -
acc: 0.9898 - val_loss: 0.0765 - val_acc: 0.9802
Epoch 8/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0244 -
```

```
acc: 0.9919 - val_loss: 0.0875 - val_acc: 0.9764
Epoch 9/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0205 -
acc: 0.9934 - val_loss: 0.0840 - val_acc: 0.9784
Epoch 10/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.0191 -
acc: 0.9937 - val_loss: 0.1042 - val_acc: 0.9761
Epoch 11/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.0200 -
acc: 0.9933 - val_loss: 0.1077 - val_acc: 0.9743
Epoch 12/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.0161 -
acc: 0.9947 - val_loss: 0.1308 - val_acc: 0.9662
Epoch 13/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.0145 -
acc: 0.9954 - val_loss: 0.1026 - val_acc: 0.9762
Epoch 14/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0125 -
acc: 0.9960 - val_loss: 0.1330 - val_acc: 0.9746
Epoch 15/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0139 -
acc: 0.9951 - val_loss: 0.1005 - val_acc: 0.9766
Epoch 16/20
60000/60000 [==============================] - 3s 43us/step - loss: 0.0105 -
acc: 0.9965 - val_loss: 0.1085 - val_acc: 0.9771
Epoch 17/20
60000/60000 [==============================] - 3s 43us/step - loss: 0.0136 -
acc: 0.9957 - val_loss: 0.1148 - val_acc: 0.9746
Epoch 18/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0118 -
acc: 0.9963 - val_loss: 0.0982 - val_acc: 0.9776
Epoch 19/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.0096 -
acc: 0.9968 - val_loss: 0.0994 - val_acc: 0.9792
Epoch 20/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0074 -
acc: 0.9976 - val_loss: 0.1014 - val_acc: 0.9809
```

In [67]:

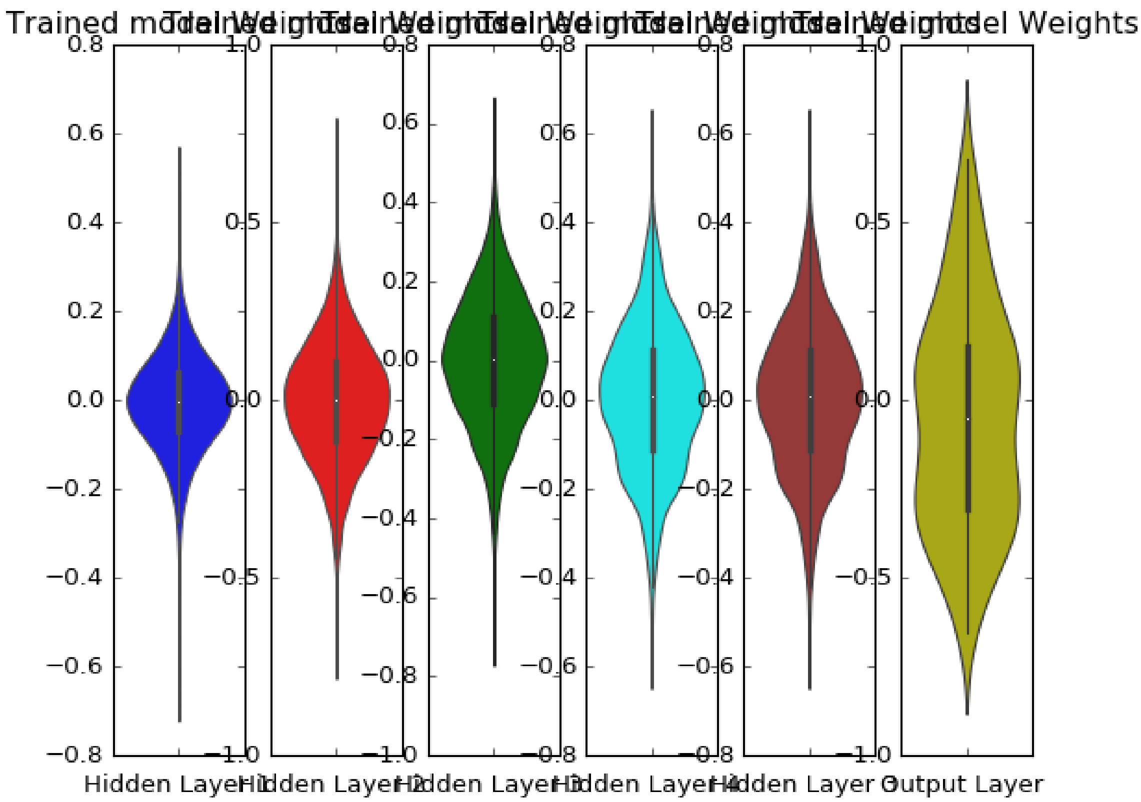
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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))

# 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, verbos

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

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


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

```
Test score: 0.10143251653150177
Test accuracy: 0.9809
```



In [68]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='brown')
plt.xlabel('Hidden Layer 3 ')


plt.subplot(1, 6, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



MLP + Batch-Norm on 5 hidden Layers + AdamOptimizer

In [69]:

```
# 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=v
# 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(256, activation='relu', input_shape=(input_dim,), kernel_initializer=
model_batch.add(BatchNormalization())

model_batch.add(Dense(132, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stc
model_batch.add(BatchNormalization())

model_batch.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdc
model_batch.add(BatchNormalization())

model_batch.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdc
model_batch.add(BatchNormalization())

model_batch.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdc
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_49 (Dense)             (None, 256)               200960    
_________________________________________________________________
batch_normalization_16 (Batc (None, 256)               1024      
_________________________________________________________________
dense_50 (Dense)             (None, 132)               33924     
_________________________________________________________________
batch_normalization_17 (Batc (None, 132)               528       
_________________________________________________________________
dense_51 (Dense)             (None, 64)                8512      
_________________________________________________________________
batch_normalization_18 (Batc (None, 64)                256       
_________________________________________________________________
dense_52 (Dense)             (None, 32)                2080      
_________________________________________________________________
batch_normalization_19 (Batc (None, 32)                128       
_________________________________________________________________
dense_53 (Dense)             (None, 16)                528       
_________________________________________________________________
batch_normalization_20 (Batc (None, 16)                64        
_________________________________________________________________
dense_54 (Dense)             (None, 10)                170       
=================================================================
Total params: 248,174
```

```
Trainable params: 247,174
Non-trainable params: 1,000
_________________________________________________________________
```

In [70]:

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

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.3940 -
acc: 0.9010 - val_loss: 0.1705 - val_acc: 0.9525
Epoch 2/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.1194 -
acc: 0.9662 - val_loss: 0.1321 - val_acc: 0.9620
Epoch 3/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0810 -
acc: 0.9761 - val_loss: 0.1025 - val_acc: 0.9690
Epoch 4/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0632 -
acc: 0.9812 - val_loss: 0.0927 - val_acc: 0.9718
Epoch 5/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0488 -
acc: 0.9851 - val_loss: 0.0833 - val_acc: 0.9759
Epoch 6/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0433 -
acc: 0.9864 - val_loss: 0.0792 - val_acc: 0.9765
Epoch 7/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0349 -
acc: 0.9892 - val_loss: 0.0881 - val_acc: 0.9750
Epoch 8/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0321 -
acc: 0.9900 - val_loss: 0.0759 - val_acc: 0.9786
Epoch 9/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0287 -
acc: 0.9909 - val_loss: 0.0840 - val_acc: 0.9771
Epoch 10/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0261 -
acc: 0.9915 - val_loss: 0.0810 - val_acc: 0.9778
Epoch 11/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0237 -
acc: 0.9920 - val_loss: 0.0764 - val_acc: 0.9797
Epoch 12/20
60000/60000 [==============================] - 5s 78us/step - loss: 0.0234 -
acc: 0.9924 - val_loss: 0.0791 - val_acc: 0.9785
Epoch 13/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0198 -
acc: 0.9934 - val_loss: 0.0763 - val_acc: 0.9801
Epoch 14/20
60000/60000 [==============================] - 5s 75us/step - loss: 0.0177 -
acc: 0.9942 - val_loss: 0.0852 - val_acc: 0.9785
Epoch 15/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0176 -
acc: 0.9940 - val_loss: 0.0802 - val_acc: 0.9781
Epoch 16/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0162 -
acc: 0.9945 - val_loss: 0.0783 - val_acc: 0.9801
Epoch 17/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0130 -
acc: 0.9955 - val_loss: 0.0743 - val_acc: 0.9811
Epoch 18/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0138 -
acc: 0.9954 - val_loss: 0.0809 - val_acc: 0.9813
```

```
Epoch 19/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.0126 -
acc: 0.9958 - val_loss: 0.0898 - val_acc: 0.9785
Epoch 20/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0119 -
acc: 0.9964 - val_loss: 0.0782 - val_acc: 0.9812
```

In [71]:

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

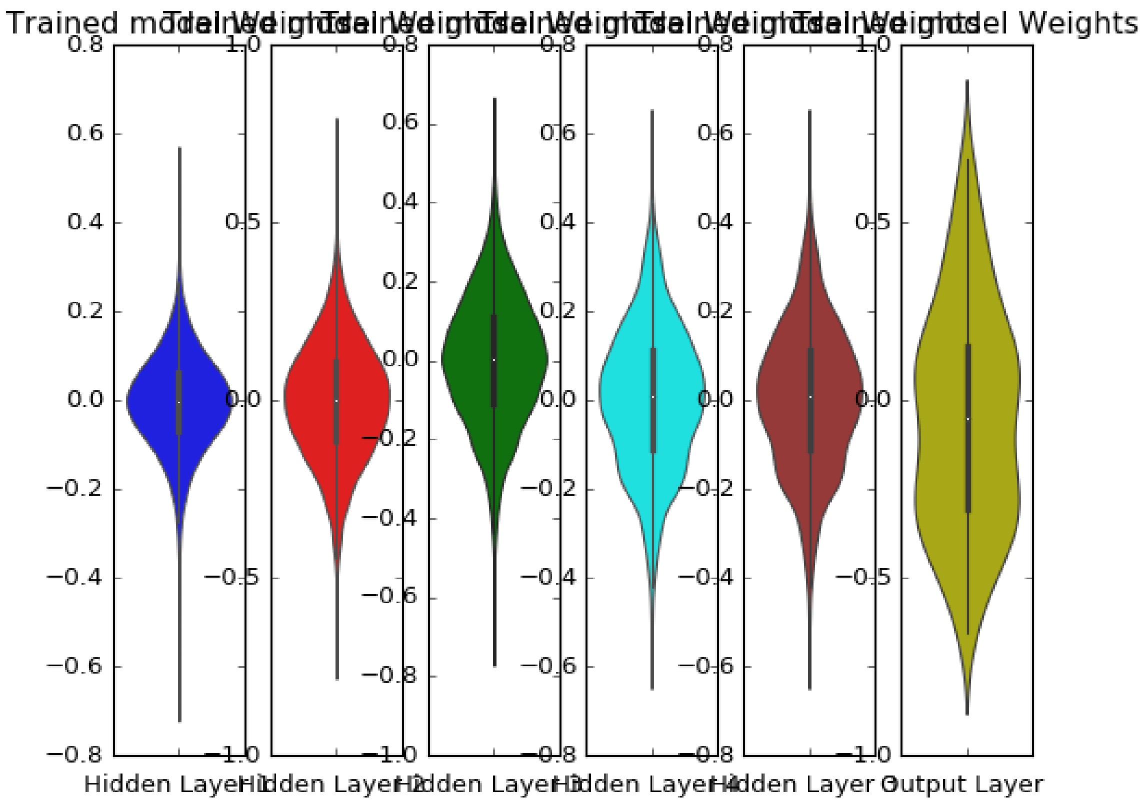
```
Test score: 0.07821065115529928
Test accuracy: 0.9812
```

In [72]:

```


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, verbos

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

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

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

In [73]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='brown')
plt.xlabel('Hidden Layer 3 ')


plt.subplot(1, 6, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



MLP + Dropout(rate = 0.5) + AdamOptimizer with 5 hidden layers

In [74]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(256, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdc
model_drop.add(Dropout(0.5))

model_drop.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.5))

model_drop.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.5))

model_drop.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_55 (Dense)             (None, 256)               200960    
_________________________________________________________________
dropout_21 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_56 (Dense)             (None, 128)               32896     
_________________________________________________________________
dropout_22 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_57 (Dense)             (None, 64)                8256      
_________________________________________________________________
dropout_23 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_58 (Dense)             (None, 32)                2080      
_________________________________________________________________
dropout_24 (Dropout)         (None, 32)                0         
_________________________________________________________________
dense_59 (Dense)             (None, 16)                528       
_________________________________________________________________
dropout_25 (Dropout)         (None, 16)                0         
_________________________________________________________________
dense_60 (Dense)             (None, 10)                170       
=================================================================
Total params: 244,890
Trainable params: 244,890
Non-trainable params: 0
_________________________________________________________________
```

In [75]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 77us/step - loss: 2.0919 -
acc: 0.2437 - val_loss: 1.4719 - val_acc: 0.5675
Epoch 2/20
60000/60000 [==============================] - 5s 76us/step - loss: 1.5121 -
acc: 0.4698 - val_loss: 1.0775 - val_acc: 0.6651
Epoch 3/20
60000/60000 [==============================] - 4s 65us/step - loss: 1.2592 -
acc: 0.5563 - val_loss: 0.9283 - val_acc: 0.7135
Epoch 4/20
60000/60000 [==============================] - 4s 60us/step - loss: 1.1274 -
acc: 0.5964 - val_loss: 0.8500 - val_acc: 0.7431
Epoch 5/20
60000/60000 [==============================] - 4s 64us/step - loss: 1.0385 -
acc: 0.6244 - val_loss: 0.7729 - val_acc: 0.7404
Epoch 6/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.9854 -
acc: 0.6372 - val_loss: 0.7186 - val_acc: 0.7485
Epoch 7/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.9358 -
acc: 0.6564 - val_loss: 0.7074 - val_acc: 0.7570
Epoch 8/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.9070 -
acc: 0.6706 - val_loss: 0.6708 - val_acc: 0.7626
Epoch 9/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.8758 -
acc: 0.6771 - val_loss: 0.6378 - val_acc: 0.7682
Epoch 10/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.8545 -
acc: 0.6867 - val_loss: 0.6129 - val_acc: 0.7741
Epoch 11/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.8247 -
acc: 0.6961 - val_loss: 0.5988 - val_acc: 0.7782
Epoch 12/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.8060 -
acc: 0.7010 - val_loss: 0.5926 - val_acc: 0.7778
Epoch 13/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.7989 -
acc: 0.7045 - val_loss: 0.5832 - val_acc: 0.7754
Epoch 14/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.7900 -
acc: 0.7084 - val_loss: 0.5665 - val_acc: 0.7786
Epoch 15/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.7791 -
acc: 0.7119 - val_loss: 0.5853 - val_acc: 0.7789
Epoch 16/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.7677 -
acc: 0.7154 - val_loss: 0.5975 - val_acc: 0.7814
Epoch 17/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.7564 -
acc: 0.7220 - val_loss: 0.5574 - val_acc: 0.7841
Epoch 18/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.7600 -
acc: 0.7284 - val_loss: 0.5639 - val_acc: 0.7812
```

```
Epoch 19/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.7450 -
acc: 0.7352 - val_loss: 0.5385 - val_acc: 0.8084
Epoch 20/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.7221 -
acc: 0.7454 - val_loss: 0.5309 - val_acc: 0.8095
```

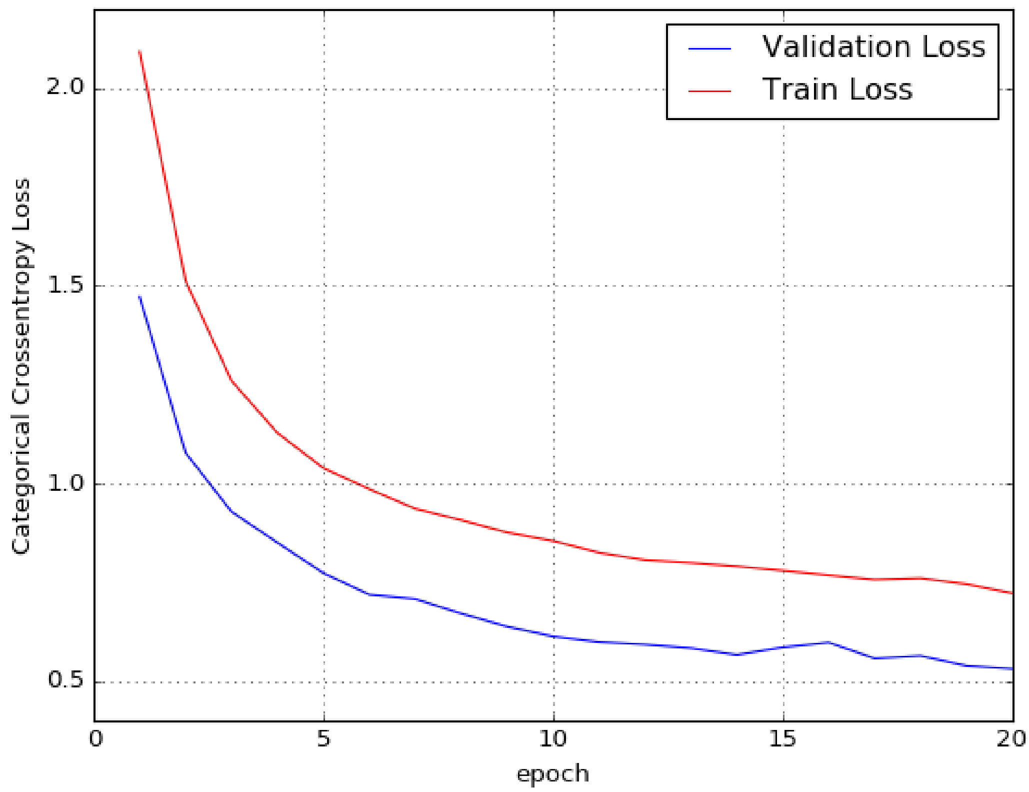
In [76]:

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

```
Test score: 0.5309434900760651
Test accuracy: 0.8095
```

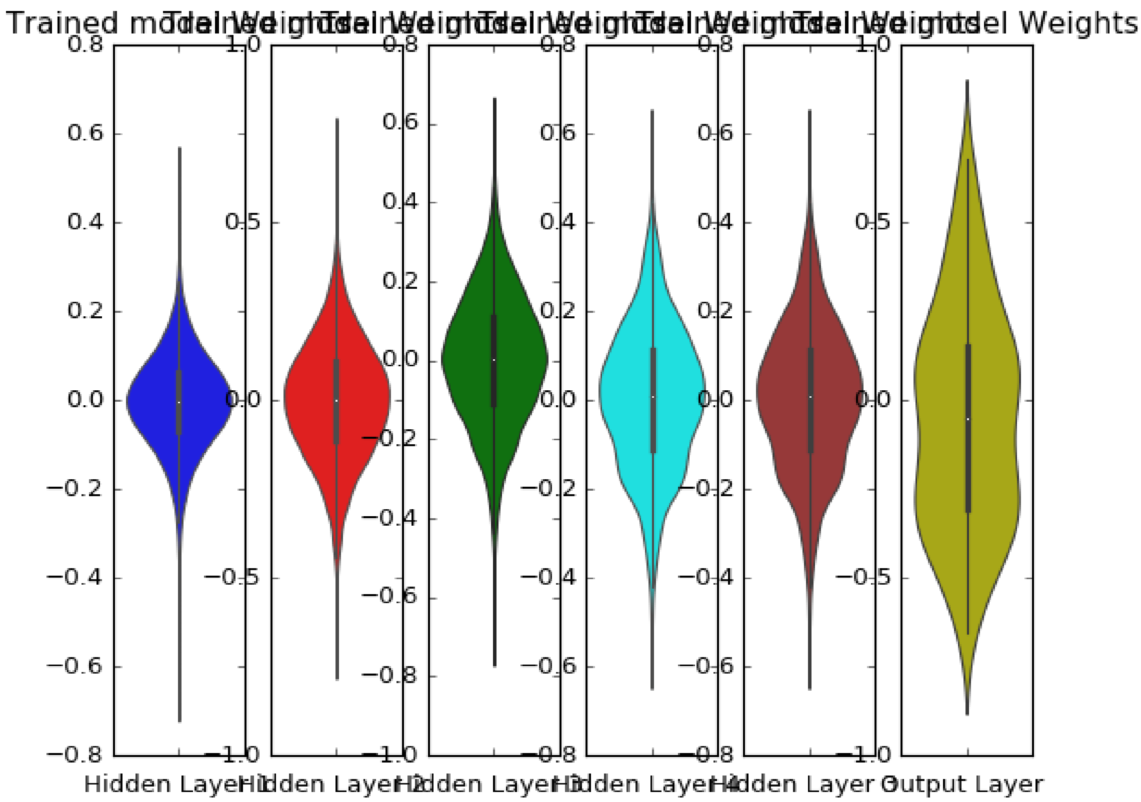
In [77]:

```
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, verbos

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

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

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



In [78]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='brown')
plt.xlabel('Hidden Layer 3 ')


plt.subplot(1, 6, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



MLP + Dropout(rate = 0.3) + AdamOptimizer with 5 hidden layers

In [79]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(256, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(Dropout(0.3))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_drop.add(Dropout(0.3))

model_drop.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.3))

model_drop.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.3))

model_drop.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.3))

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_61 (Dense)             (None, 256)               200960    
_________________________________________________________________
dropout_26 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_62 (Dense)             (None, 128)               32896     
_________________________________________________________________
dropout_27 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_63 (Dense)             (None, 64)                8256      
_________________________________________________________________
dropout_28 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_64 (Dense)             (None, 32)                2080      
_________________________________________________________________
dropout_29 (Dropout)         (None, 32)                0         
_________________________________________________________________
dense_65 (Dense)             (None, 16)                528       
_________________________________________________________________
dropout_30 (Dropout)         (None, 16)                0         
_________________________________________________________________
dense_66 (Dense)             (None, 10)                170       
=================================================================
Total params: 244,890
Trainable params: 244,890
Non-trainable params: 0
_________________________________________________________________
```

In [80]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 79us/step - loss: 1.2953 -
acc: 0.5550 - val_loss: 0.3444 - val_acc: 0.9196
Epoch 2/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.6133 -
acc: 0.8158 - val_loss: 0.2037 - val_acc: 0.9518
Epoch 3/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.4686 -
acc: 0.8674 - val_loss: 0.1706 - val_acc: 0.9584
Epoch 4/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.3907 -
acc: 0.8955 - val_loss: 0.1666 - val_acc: 0.9600
Epoch 5/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.3488 -
acc: 0.9061 - val_loss: 0.1574 - val_acc: 0.9633
Epoch 6/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.3226 -
acc: 0.9148 - val_loss: 0.1406 - val_acc: 0.9700
Epoch 7/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.2941 -
acc: 0.9220 - val_loss: 0.1450 - val_acc: 0.9679
Epoch 8/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.2711 -
acc: 0.9304 - val_loss: 0.1344 - val_acc: 0.9682
Epoch 9/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.2582 -
acc: 0.9318 - val_loss: 0.1334 - val_acc: 0.9721
Epoch 10/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.2399 -
acc: 0.9374 - val_loss: 0.1381 - val_acc: 0.9713
Epoch 11/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.2325 -
acc: 0.9389 - val_loss: 0.1267 - val_acc: 0.9736
Epoch 12/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.2199 -
acc: 0.9426 - val_loss: 0.1407 - val_acc: 0.9708
Epoch 13/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.2073 -
acc: 0.9456 - val_loss: 0.1249 - val_acc: 0.9734
Epoch 14/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.2039 -
acc: 0.9464 - val_loss: 0.1340 - val_acc: 0.9731
Epoch 15/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.2019 -
acc: 0.9470 - val_loss: 0.1302 - val_acc: 0.9746
Epoch 16/20
60000/60000 [==============================] - 5s 75us/step - loss: 0.1932 -
acc: 0.9503 - val_loss: 0.1149 - val_acc: 0.9756
Epoch 17/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.1823 -
acc: 0.9527 - val_loss: 0.1123 - val_acc: 0.9785
Epoch 18/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.1789 -
acc: 0.9540 - val_loss: 0.1136 - val_acc: 0.9766
```

```
Epoch 19/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1710 -
acc: 0.9562 - val_loss: 0.1113 - val_acc: 0.9788
Epoch 20/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.1696 -
acc: 0.9562 - val_loss: 0.1158 - val_acc: 0.9794
```

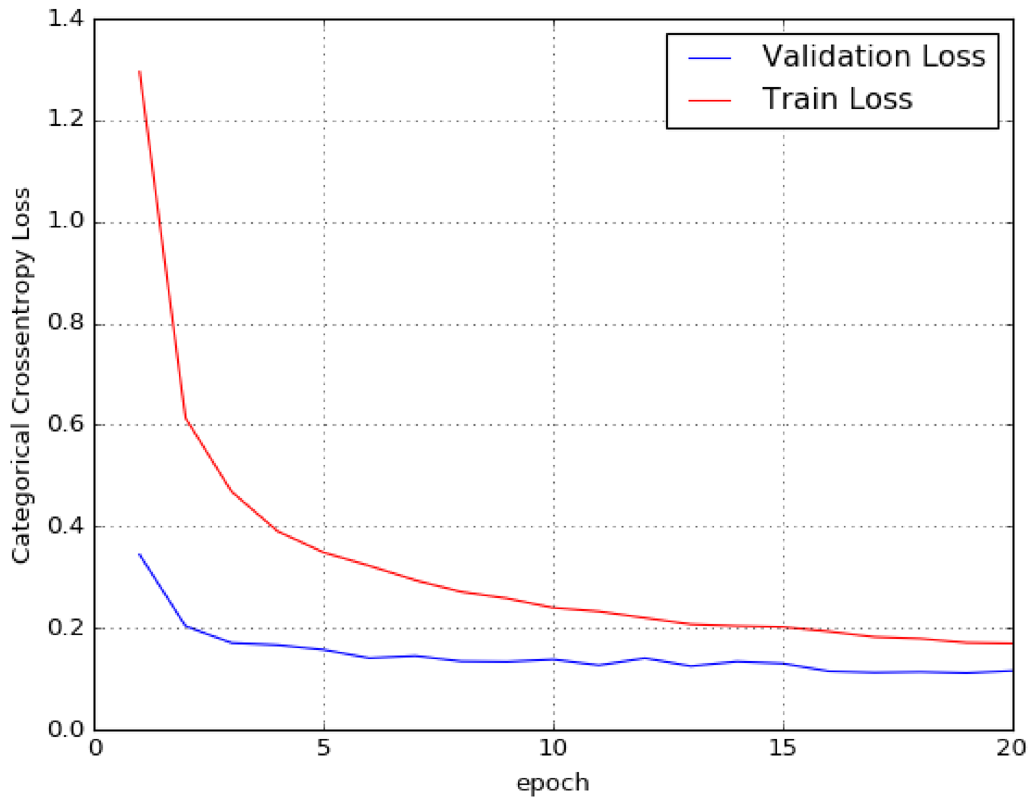
In [81]:

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

```
Test score: 0.1157619362520054
Test accuracy: 0.9794
```

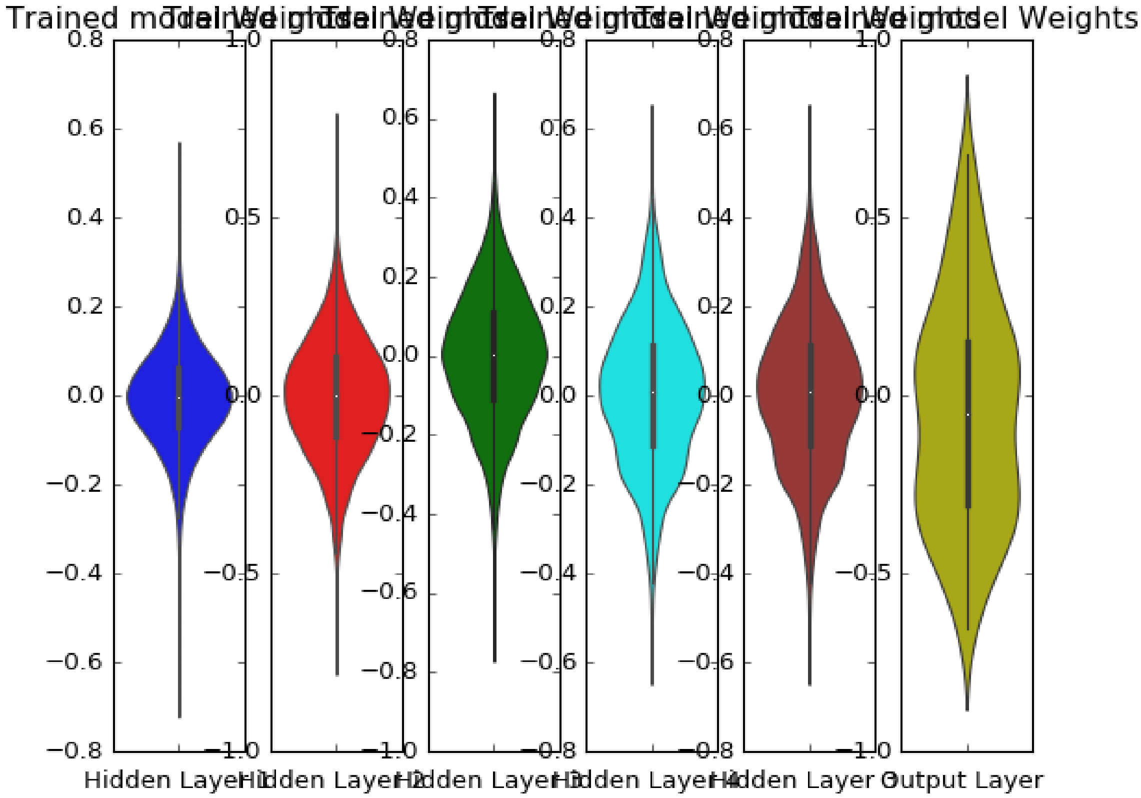
In [82]:

```
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, verbos

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

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

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



In [83]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='brown')
plt.xlabel('Hidden Layer 3 ')


plt.subplot(1, 6, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



MLP + BatchNormalization + Dropout(rate = 0.5) + AdamOptimizer with 3 hidden layers

In [84]:

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

from keras.layers import Dropout

model_drop = Sequential()

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

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
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_67 (Dense)             (None, 256)               200960    
_________________________________________________________________
batch_normalization_21 (Batc (None, 256)               1024      
_________________________________________________________________
dropout_31 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_68 (Dense)             (None, 128)               32896     
_________________________________________________________________
batch_normalization_22 (Batc (None, 128)               512       
_________________________________________________________________
dropout_32 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_69 (Dense)             (None, 64)                8256      
_________________________________________________________________
batch_normalization_23 (Batc (None, 64)                256       
_________________________________________________________________
dropout_33 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_70 (Dense)             (None, 32)                2080      
_________________________________________________________________
batch_normalization_24 (Batc (None, 32)                128       
_________________________________________________________________
dropout_34 (Dropout)         (None, 32)                0         
_________________________________________________________________
```

```
dense_71 (Dense)             (None, 16)                528
_________________________________________________________________
batch_normalization_25 (Batc (None, 16)                64
_________________________________________________________________
dropout_35 (Dropout)         (None, 16)                0
_________________________________________________________________
dense_72 (Dense)             (None, 10)                170
=================================================================
Total params: 246,874
Trainable params: 245,882
Non-trainable params: 992
_________________________________________________________________
```

In [85]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 121us/step - loss: 2.2357
- acc: 0.2311 - val_loss: 1.3396 - val_acc: 0.7074
Epoch 2/20
60000/60000 [==============================] - 5s 83us/step - loss: 1.5868 -
acc: 0.4290 - val_loss: 0.8430 - val_acc: 0.7752
Epoch 3/20
60000/60000 [==============================] - 5s 83us/step - loss: 1.2666 -
acc: 0.5369 - val_loss: 0.6421 - val_acc: 0.8368
Epoch 4/20
60000/60000 [==============================] - 5s 87us/step - loss: 1.0828 -
acc: 0.6060 - val_loss: 0.5345 - val_acc: 0.8755
Epoch 5/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.9650 -
acc: 0.6517 - val_loss: 0.4411 - val_acc: 0.9000
Epoch 6/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.8813 -
acc: 0.6906 - val_loss: 0.3809 - val_acc: 0.9190
Epoch 7/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.7947 -
acc: 0.7292 - val_loss: 0.3086 - val_acc: 0.9374
Epoch 8/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.7309 -
acc: 0.7568 - val_loss: 0.2643 - val_acc: 0.9387
Epoch 9/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.6728 -
acc: 0.7874 - val_loss: 0.2188 - val_acc: 0.9487
Epoch 10/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.6153 -
acc: 0.8104 - val_loss: 0.2023 - val_acc: 0.9529
Epoch 11/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.5722 -
acc: 0.8287 - val_loss: 0.1780 - val_acc: 0.9557
Epoch 12/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.5389 -
acc: 0.8404 - val_loss: 0.1658 - val_acc: 0.9583
Epoch 13/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.5108 -
acc: 0.8528 - val_loss: 0.1583 - val_acc: 0.9609
Epoch 14/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.4867 -
acc: 0.8607 - val_loss: 0.1645 - val_acc: 0.9599
Epoch 15/20
60000/60000 [==============================] - 5s 92us/step - loss: 0.4716 -
acc: 0.8661 - val_loss: 0.1575 - val_acc: 0.9626
Epoch 16/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.4561 -
acc: 0.8706 - val_loss: 0.1447 - val_acc: 0.9651
Epoch 17/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.4360 -
acc: 0.8786 - val_loss: 0.1449 - val_acc: 0.9658
Epoch 18/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.4191 -
acc: 0.8827 - val_loss: 0.1466 - val_acc: 0.9661
```

```
Epoch 19/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.4146 -
acc: 0.8872 - val_loss: 0.1389 - val_acc: 0.9674
Epoch 20/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.3964 -
acc: 0.8903 - val_loss: 0.1381 - val_acc: 0.9688
```

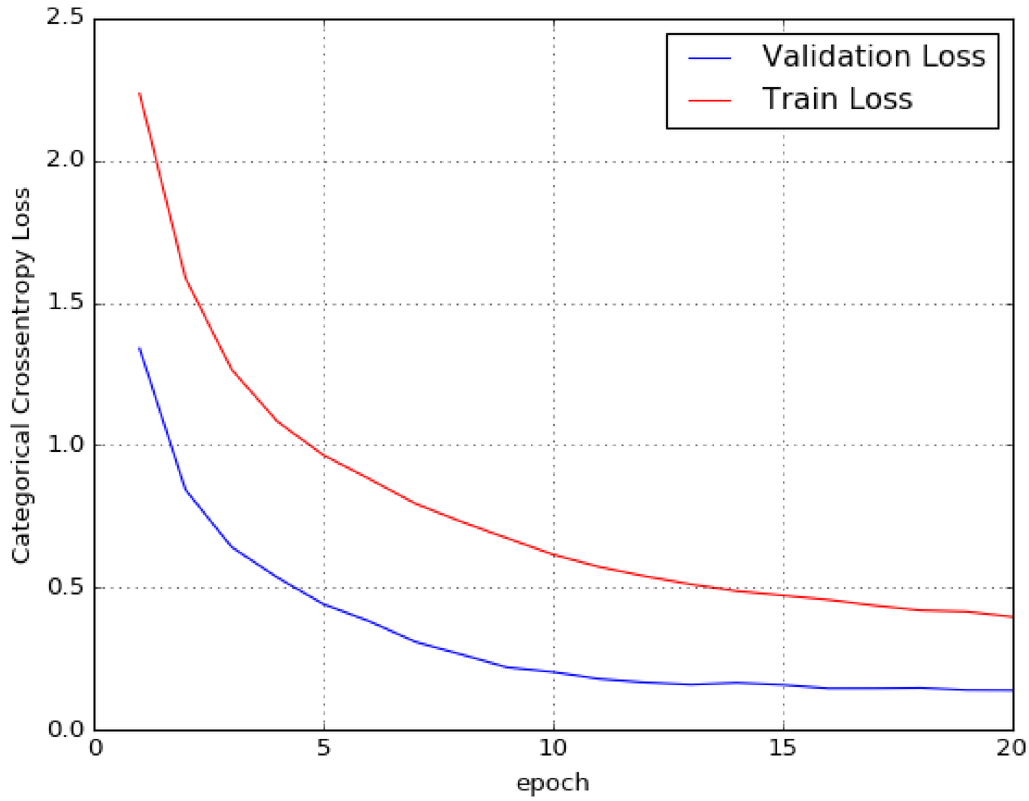
In [86]:

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

```
Test score: 0.13805706812888383
Test accuracy: 0.9688
```

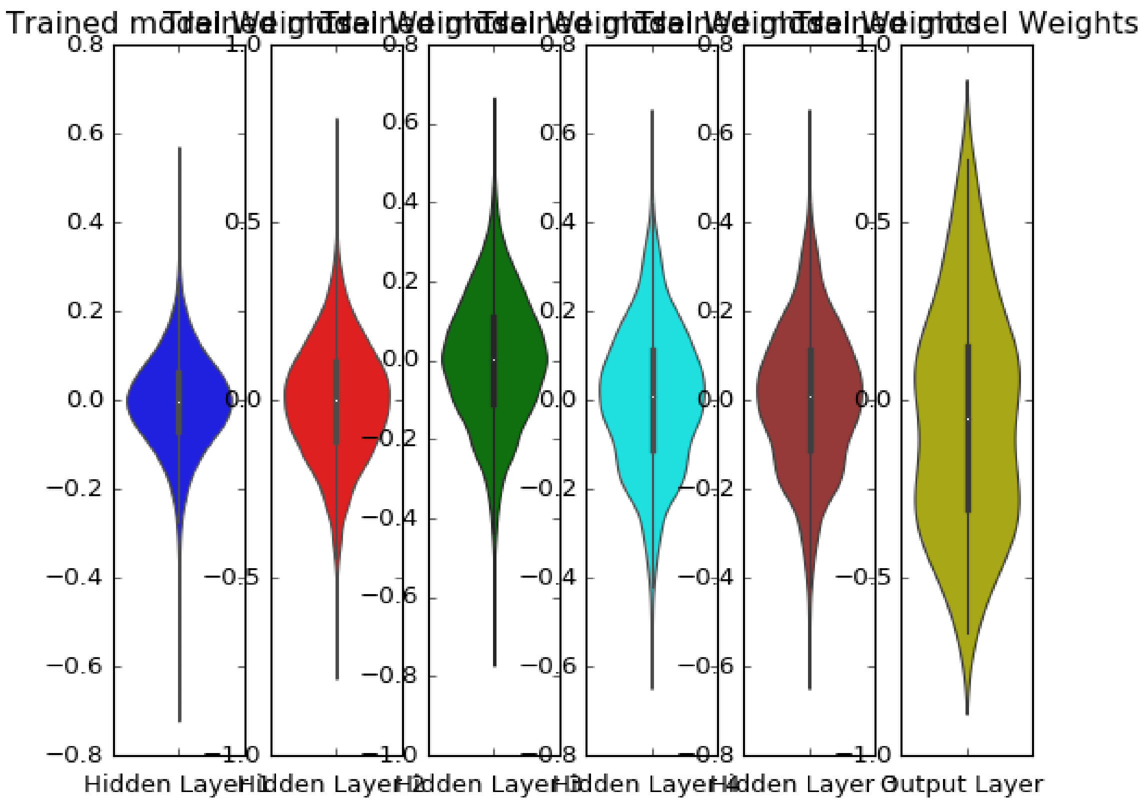
In [87]:

```
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, verbos

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

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

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



In [88]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='brown')
plt.xlabel('Hidden Layer 3 ')


plt.subplot(1, 6, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



MLP + BatchNormalization + Dropout(rate = 0.3) + AdamOptimizer with 3 hidden layers

In [89]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(256, activation='relu', input_shape=(input_dim,), kernel_initializer=R
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

model_drop.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

model_drop.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

model_drop.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

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


model_drop.summary()
```

Layer (type)	Output Shape	Param #
dense_73 (Dense)	(None, 256)	200960
batch_normalization_26 (Batc	(None, 256)	1024
dropout_36 (Dropout)	(None, 256)	0
dense_74 (Dense)	(None, 128)	32896
batch_normalization_27 (Batc	(None, 128)	512
dropout_37 (Dropout)	(None, 128)	0
dense_75 (Dense)	(None, 64)	8256
batch_normalization_28 (Batc	(None, 64)	256
dropout_38 (Dropout)	(None, 64)	0
dense_76 (Dense)	(None, 32)	2080
batch_normalization_29 (Batc	(None, 32)	128
dropout_39 (Dropout)	(None, 32)	0

```
dense_77 (Dense)             (None, 16)                528
_________________________________________________________________
batch_normalization_30 (Batc (None, 16)                64
_________________________________________________________________
dropout_40 (Dropout)         (None, 16)                0
_________________________________________________________________
dense_78 (Dense)             (None, 10)                170
=================================================================
Total params: 246,874
Trainable params: 245,882
Non-trainable params: 992
_________________________________________________________________
```

In [90]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 127us/step - loss: 1.3488
- acc: 0.5626 - val_loss: 0.3823 - val_acc: 0.9094
Epoch 2/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.6504 -
acc: 0.8141 - val_loss: 0.2155 - val_acc: 0.9436
Epoch 3/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.4596 -
acc: 0.8756 - val_loss: 0.1916 - val_acc: 0.9499
Epoch 4/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.3727 -
acc: 0.9042 - val_loss: 0.1556 - val_acc: 0.9577
Epoch 5/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.3184 -
acc: 0.9225 - val_loss: 0.1419 - val_acc: 0.9614
Epoch 6/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.2840 -
acc: 0.9301 - val_loss: 0.1332 - val_acc: 0.9652
Epoch 7/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.2605 -
acc: 0.9364 - val_loss: 0.1287 - val_acc: 0.9680
Epoch 8/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.2333 -
acc: 0.9447 - val_loss: 0.1204 - val_acc: 0.9717
Epoch 9/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.2152 -
acc: 0.9466 - val_loss: 0.1132 - val_acc: 0.9721
Epoch 10/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.2051 -
acc: 0.9512 - val_loss: 0.1213 - val_acc: 0.9713
Epoch 11/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.1973 -
acc: 0.9532 - val_loss: 0.1044 - val_acc: 0.9735
Epoch 12/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1844 -
acc: 0.9558 - val_loss: 0.1013 - val_acc: 0.9764
Epoch 13/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.1732 -
acc: 0.9595 - val_loss: 0.0936 - val_acc: 0.9770
Epoch 14/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.1682 -
acc: 0.9610 - val_loss: 0.0920 - val_acc: 0.9782
Epoch 15/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1559 -
acc: 0.9627 - val_loss: 0.0934 - val_acc: 0.9777
Epoch 16/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.1485
- acc: 0.9651 - val_loss: 0.0998 - val_acc: 0.9765
Epoch 17/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.1447 -
acc: 0.9668 - val_loss: 0.1004 - val_acc: 0.9765
Epoch 18/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1361 -
acc: 0.9673 - val_loss: 0.0935 - val_acc: 0.9793
```

```
Epoch 19/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.1354 -
acc: 0.9694 - val_loss: 0.0916 - val_acc: 0.9789
Epoch 20/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.1322
- acc: 0.9685 - val_loss: 0.0917 - val_acc: 0.9776
```

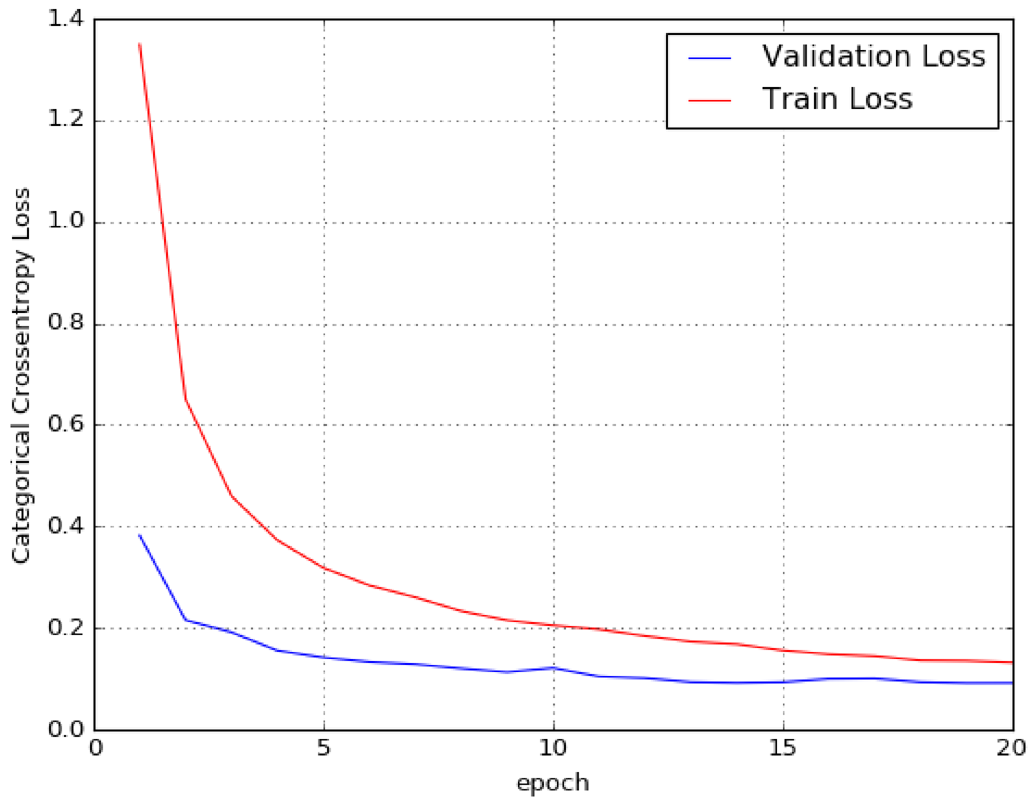
In [91]:

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

```
Test score: 0.09172347799413838
Test accuracy: 0.9776
```

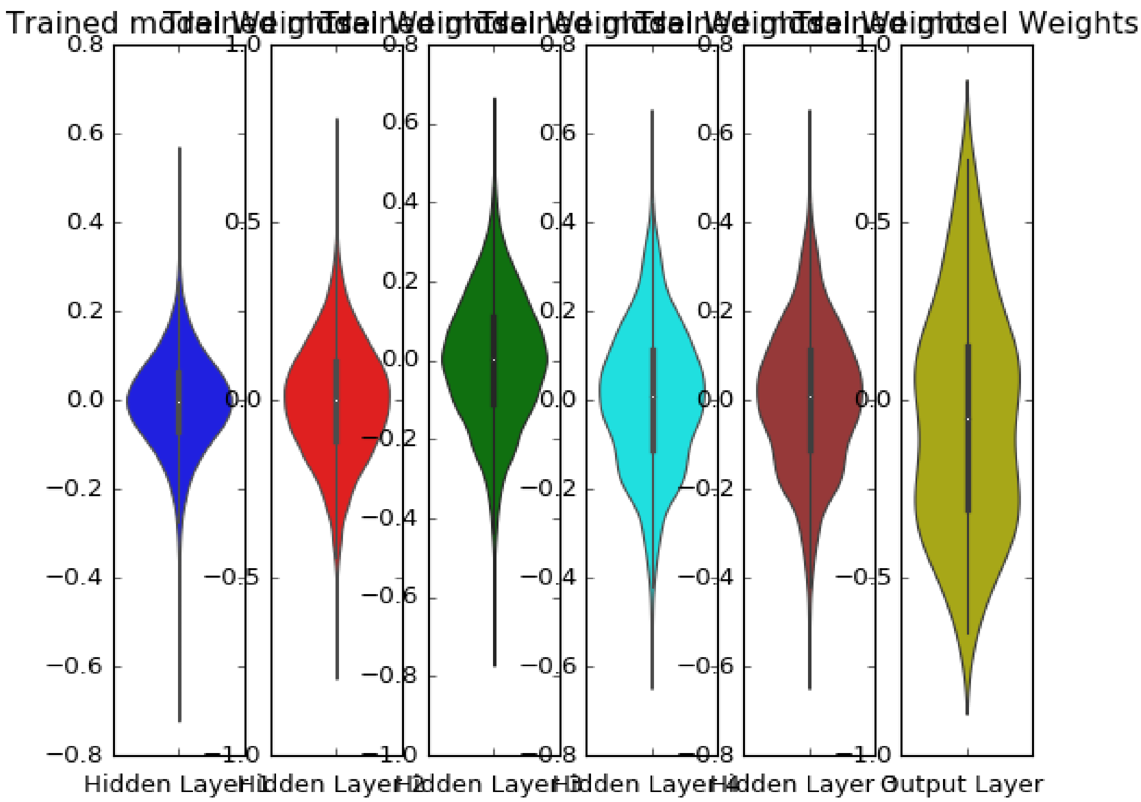
In [92]:

```
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, verbos

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

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

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



In [93]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='brown')
plt.xlabel('Hidden Layer 3 ')


plt.subplot(1, 6, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



Comparisons

In [100]:

```
from prettytable import PrettyTable

x = PrettyTable()
x.field_names = ["Architecture", "parameters", "Accuracy"]

x.add_row(["2 layer", "without Dropout and Batch Normalization", 97.9])
x.add_row(["2 layer", "with Dropuot(rate = 0.5)", 97.44])
x.add_row(["2 layer", "with Dropuot(rate = 0.3)", 98.02])
x.add_row(["2 layer", "with Batch Normalization", 97.98])
x.add_row(["2 layer", "with Dropuot(rate = 0.5) and Batch Normalization ", 98.08])
x.add_row(["2 layer", "with Dropuot(rate = 0.3) and Batch Normalization ", 98.25])

x.add_row(["3 layer", "without Dropout and Batch Normalization", 97.87])
x.add_row(["3 layer", "with Dropuot", 97.38])
x.add_row(["3 layer", "with Dropuot(rate = 0.3)", 98.06])
x.add_row(["3 layer", "with Batch Normalization", 97.59])
x.add_row(["3 layer", "with Dropuot(rate = 0.5) and Batch Normalization ", 97.84])
x.add_row(["3 layer", "with Dropuot(rate = 0.3) and Batch Normalization ", 98.35])

x.add_row(["5 layer", "without Dropout and Batch Normalization", 98.09])
x.add_row(["5 layer", "with Dropuot(rate = 0.5)", 85.8])
x.add_row(["5 layer", "with Dropuot(rate = 0.3)", 97.54])
x.add_row(["5 layer", "with Batch Normalization", 98.14])
x.add_row(["5 layer", "with Dropuot(rate = 0.5) and Batch Normalization ", 95.9])
x.add_row(["5 layer", "with Dropuot(rate = 0.3) and Batch Normalization ", 97.76])

print(x)
```

```
+--------------+----------------------------------------------------+--------
--+
| Architecture |                     parameters                     | Accurac
y |
+--------------+----------------------------------------------------+--------
--+
|   2 layer    |      without Dropout and Batch Normalization       |   97.9
|
|   2 layer    |              with Dropuot(rate = 0.5)              |  97.44
|
|   2 layer    |              with Dropuot(rate = 0.3)              |  98.02
|
|   2 layer    |              with Batch Normalization              |  97.98
|
|   2 layer    | with Dropuot(rate = 0.5) and Batch Normalization   |  98.08
|
|   2 layer    | with Dropuot(rate = 0.3) and Batch Normalization   |  98.25
|
|   3 layer    |      without Dropout and Batch Normalization       |  97.87
|
|   3 layer    |                    with Dropuot                    |  97.38
|
|   3 layer    |              with Dropuot(rate = 0.3)              |  98.06
|
|   3 layer    |              with Batch Normalization              |  97.59
|
|   3 layer    | with Dropuot(rate = 0.5) and Batch Normalization   |  97.84
|
|   3 layer    | with Dropuot(rate = 0.3) and Batch Normalization   |  98.35
|
|   5 layer    |      without Dropout and Batch Normalization       |  98.09
```

```
|
|   5 layer    |              with Dropuot(rate = 0.5)             |   85.8
|
|   5 layer    |              with Dropuot(rate = 0.3)             |  97.54
|
|   5 layer    |              with Batch Normalization             |  98.14
|
|   5 layer    | with Dropuot(rate = 0.5) and Batch Normalization  |   95.9
|
|   5 layer    | with Dropuot(rate = 0.3) and Batch Normalization  |  97.76
|
+--------------+---------------------------------------------------+--------
--+
```

Procedure followed

1. Flattened the 28*28 dimensional MNIST data to 784
2. Normalized the data
3. Used a softmax classifier of output dimensions = 10
4. Created multiple models in Keras with various parameter combinations like activation function = 'relu', optimizer = 'Adam', with/without dropout of different rates, with/without Batch normalization
5. Plotted the epoch vs Train/Test loss of each model
6. Plotted the weights of each model