Keras -- MLPs on MNIST

In [2]:
```
# if you keras is not using tensorflow as backend set "KERAS_BACKEND=tensorflow" 
from keras.utils import np_utils
from keras.datasets import mnist
import seaborn as sns
from keras.initializers import RandomNormal
```

```
Using TensorFlow backend.
```

In [3]:
```
%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 [4]:
```
# 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 (https://
s3.amazonaws.com/img-datasets/mnist.npz)
11493376/11490434 [==============================] - 5s 0us/step
```

In [5]:
```
print("Number of training examples :", X_train.shape[0], "and each image is of sh
print("Number of training examples :", X_test.shape[0], "and each image is of sha
```

```
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 [6]:
```
# 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 [7]:
```
# after converting the input images from 3d to 2d vectors

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

```
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 [8]:
```
# 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 [9]:
```
# 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 d
# X => (X - Xmin)/(Xmax-Xmin) = X/255

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

In [10]:

```
# 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 [11]:

```
# 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 [12]:
```
# 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 c

# 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='g
# bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activ
# kernel_constraint=None, bias_constraint=None)

# Dense implements the operation: output = activation(dot(input, kernel) + bias) w
# activation is the element-wise activation function passed as the activation argu
# 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 acti

# 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 [18]:
```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [19]:
```
print(X_train.shape[1])
```

```
784
```

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

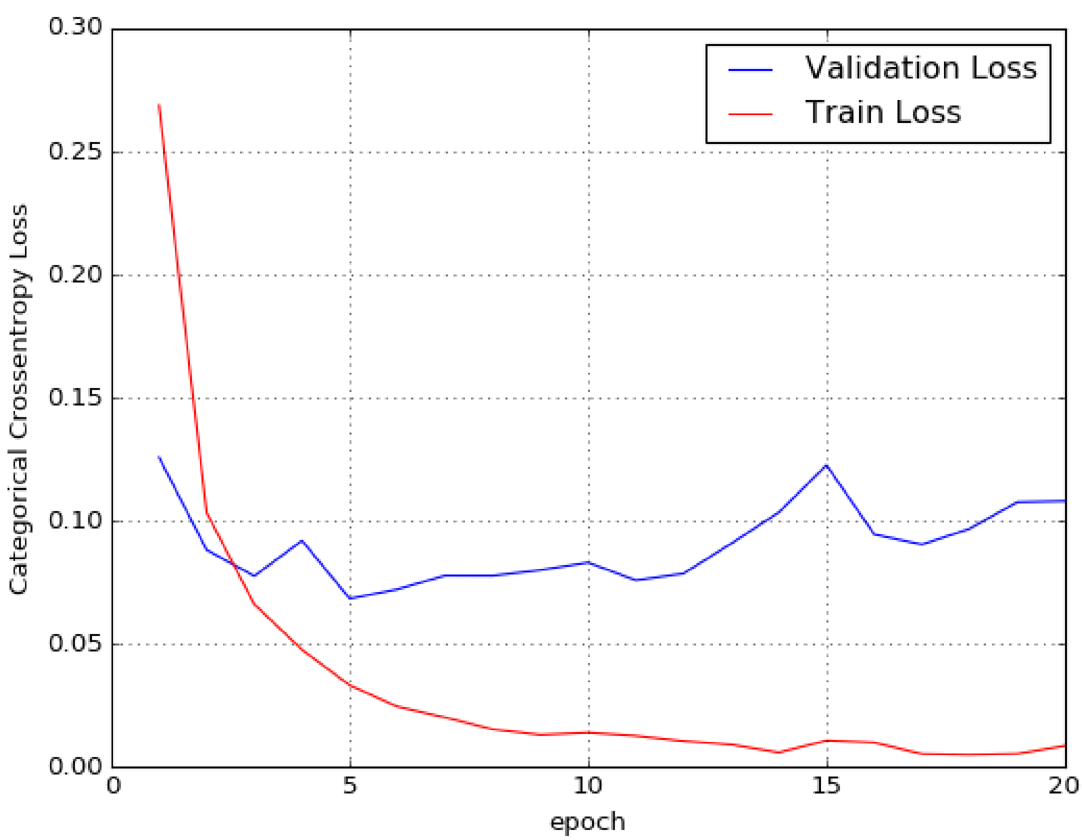
In [20]:
```
model_relu = Sequential()
model_relu.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_ini
model_relu.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_4 (Dense)              (None, 364)               285740    
_________________________________________________________________
dense_5 (Dense)              (None, 52)                18980     
_________________________________________________________________
dense_6 (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 [==============================] - 11s 180us/step - loss: 0.2688 -
acc: 0.9216 - val_loss: 0.1258 - val_acc: 0.9609
Epoch 2/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.1032 - a
cc: 0.9697 - val_loss: 0.0881 - val_acc: 0.9721
Epoch 3/20
60000/60000 [==============================] - 8s 126us/step - loss: 0.0660 - a
cc: 0.9800 - val_loss: 0.0776 - val_acc: 0.9750
Epoch 4/20
60000/60000 [==============================] - 8s 128us/step - loss: 0.0476 - a
cc: 0.9859 - val_loss: 0.0919 - val_acc: 0.9724
Epoch 5/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.0331 - a
cc: 0.9893 - val_loss: 0.0683 - val_acc: 0.9802
Epoch 6/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.0244 - a
cc: 0.9926 - val_loss: 0.0721 - val_acc: 0.9792
Epoch 7/20
60000/60000 [==============================] - 8s 140us/step - loss: 0.0201 - a
cc: 0.9938 - val_loss: 0.0777 - val_acc: 0.9787
Epoch 8/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0152 - a
cc: 0.9957 - val_loss: 0.0777 - val_acc: 0.9803
Epoch 9/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.0130 - a
cc: 0.9958 - val_loss: 0.0799 - val_acc: 0.9778
Epoch 10/20
60000/60000 [==============================] - 10s 170us/step - loss: 0.0138 -
acc: 0.9954 - val_loss: 0.0830 - val_acc: 0.9779
```

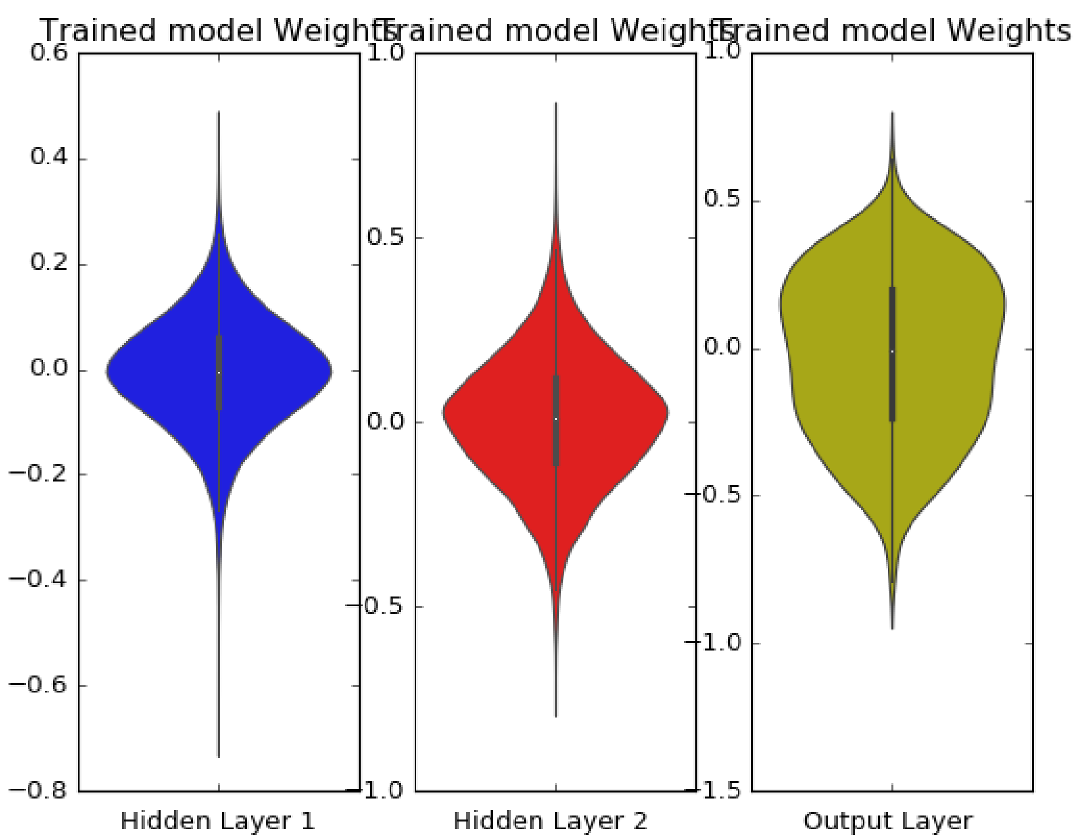
```
Epoch 11/20
60000/60000 [==============================] - 10s 165us/step - loss: 0.0126 -
acc: 0.9956 - val_loss: 0.0758 - val_acc: 0.9814
Epoch 12/20
60000/60000 [==============================] - 10s 168us/step - loss: 0.0104 -
acc: 0.9967 - val_loss: 0.0785 - val_acc: 0.9816
Epoch 13/20
60000/60000 [==============================] - 10s 161us/step - loss: 0.0090 -
acc: 0.9970 - val_loss: 0.0907 - val_acc: 0.9801
Epoch 14/20
60000/60000 [==============================] - 10s 160us/step - loss: 0.0059 -
acc: 0.9984 - val_loss: 0.1034 - val_acc: 0.9770
Epoch 15/20
60000/60000 [==============================] - 10s 162us/step - loss: 0.0106 -
acc: 0.9966 - val_loss: 0.1226 - val_acc: 0.9741
Epoch 16/20
60000/60000 [==============================] - 10s 163us/step - loss: 0.0099 -
acc: 0.9965 - val_loss: 0.0945 - val_acc: 0.9796
Epoch 17/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.0052 - a
cc: 0.9983 - val_loss: 0.0903 - val_acc: 0.9823
Epoch 18/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.0049 -
acc: 0.9986 - val_loss: 0.0967 - val_acc: 0.9813
Epoch 19/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.0053 - a
cc: 0.9985 - val_loss: 0.1076 - val_acc: 0.9792
Epoch 20/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.0085 -
acc: 0.9971 - val_loss: 0.1080 - val_acc: 0.9790
```

In [21]:
```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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


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

```
Test score: 0.10797379065210243
Test accuracy: 0.979
```



In [23]:
```
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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
```

```
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + Batch-Norm on 2 hidden Layers + AdamOptimizer

In [26]:

```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this conditio
# 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_in
model_batch.add(BatchNormalization())

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

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_11 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_3 (Batch (None, 364)               1456      
_________________________________________________________________
dense_12 (Dense)             (None, 52)                18980     
_________________________________________________________________
batch_normalization_4 (Batch (None, 52)                208       
_________________________________________________________________
dense_13 (Dense)             (None, 10)                530       
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

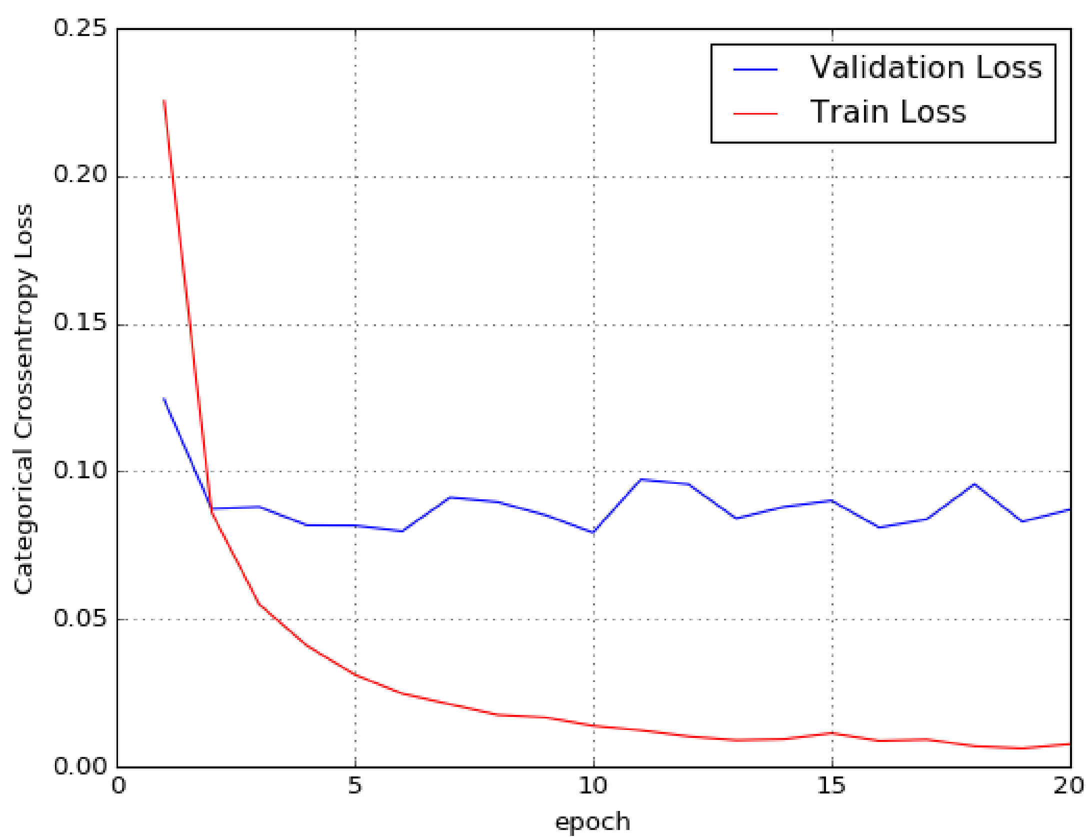
In [27]:
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 13s 219us/step - loss: 0.2254 -
acc: 0.9364 - val_loss: 0.1244 - val_acc: 0.9634
Epoch 2/20
60000/60000 [==============================] - 10s 163us/step - loss: 0.0861 -
acc: 0.9747 - val_loss: 0.0873 - val_acc: 0.9741
Epoch 3/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0549 -
acc: 0.9836 - val_loss: 0.0879 - val_acc: 0.9731
Epoch 4/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.0409 - a
cc: 0.9876 - val_loss: 0.0817 - val_acc: 0.9756
Epoch 5/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.0312 - a
cc: 0.9903 - val_loss: 0.0816 - val_acc: 0.9761
Epoch 6/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.0247 - a
cc: 0.9924 - val_loss: 0.0798 - val_acc: 0.9749
Epoch 7/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.0211 - a
cc: 0.9933 - val_loss: 0.0911 - val_acc: 0.9756
Epoch 8/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.0176 - a
cc: 0.9941 - val_loss: 0.0896 - val_acc: 0.9746
Epoch 9/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0167 - a
cc: 0.9949 - val_loss: 0.0852 - val_acc: 0.9769
Epoch 10/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.0139 - a
cc: 0.9955 - val_loss: 0.0793 - val_acc: 0.9768
Epoch 11/20
60000/60000 [==============================] - 8s 140us/step - loss: 0.0123 - a
cc: 0.9960 - val_loss: 0.0973 - val_acc: 0.9749
Epoch 12/20
60000/60000 [==============================] - 8s 140us/step - loss: 0.0103 - a
cc: 0.9970 - val_loss: 0.0956 - val_acc: 0.9766
Epoch 13/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.0090 - a
cc: 0.9972 - val_loss: 0.0840 - val_acc: 0.9782
Epoch 14/20
60000/60000 [==============================] - 11s 183us/step - loss: 0.0094 -
acc: 0.9969 - val_loss: 0.0880 - val_acc: 0.9771
Epoch 15/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.0113 -
acc: 0.9960 - val_loss: 0.0901 - val_acc: 0.9781
Epoch 16/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.0088 -
acc: 0.9976 - val_loss: 0.0810 - val_acc: 0.9780
Epoch 17/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.0092 -
acc: 0.9970 - val_loss: 0.0838 - val_acc: 0.9776
Epoch 18/20
```

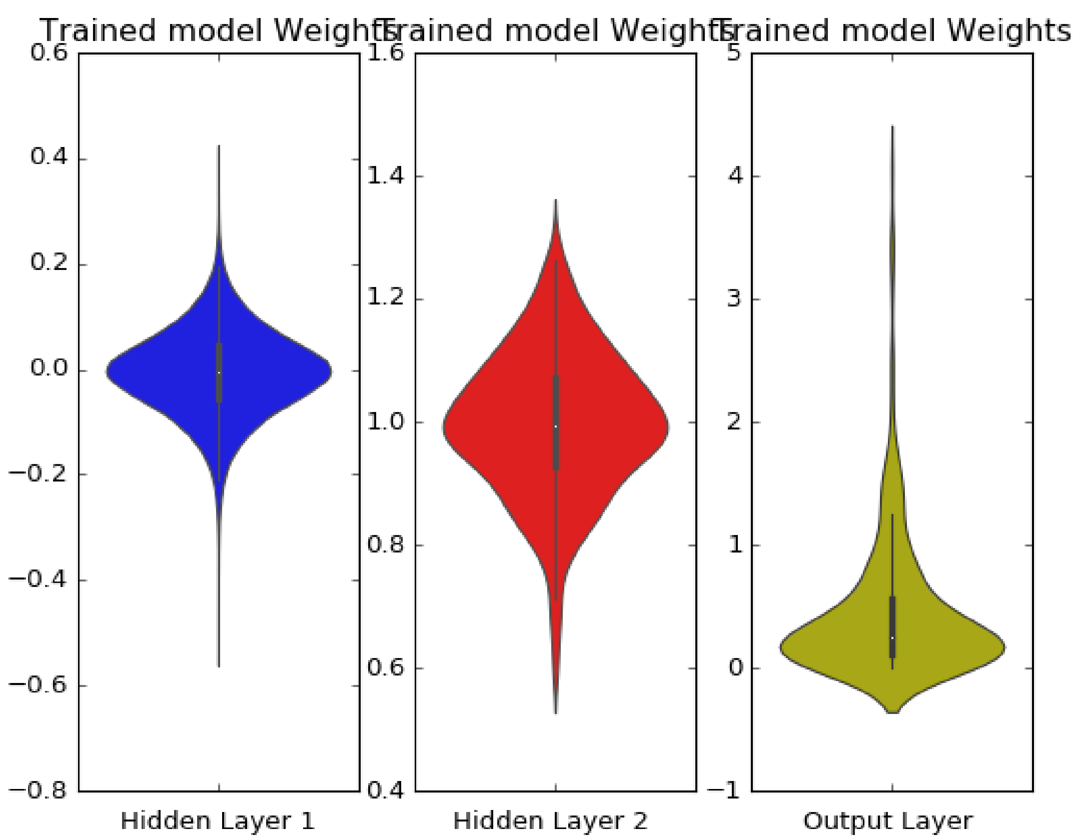
```
60000/60000 [==============================] - 11s 183us/step - loss: 0.0069 -
acc: 0.9979 - val_loss: 0.0957 - val_acc: 0.9776
Epoch 19/20
60000/60000 [==============================] - 11s 187us/step - loss: 0.0062 -
acc: 0.9980 - val_loss: 0.0830 - val_acc: 0.9778
Epoch 20/20
60000/60000 [==============================] - 11s 183us/step - loss: 0.0076 -
acc: 0.9974 - val_loss: 0.0870 - val_acc: 0.9798
```

In [28]:
```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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

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

```
Test score: 0.08703972299850357
Test accuracy: 0.9798
```



In [29]:
```
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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
```

```
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + Dropout + AdamOptimizer with 2 hidden layers

In [35]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormaliza

from keras.layers import Dropout

model_drop = Sequential()

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

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

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_20 (Dense)             (None, 364)               285740    
_________________________________________________________________
dropout_5 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_21 (Dense)             (None, 52)                18980     
_________________________________________________________________
dropout_6 (Dropout)          (None, 52)                0         
_________________________________________________________________
dense_22 (Dense)             (None, 10)                530       
=================================================================
Total params: 305,250
Trainable params: 305,250
Non-trainable params: 0
_________________________________________________________________
```

In [36]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

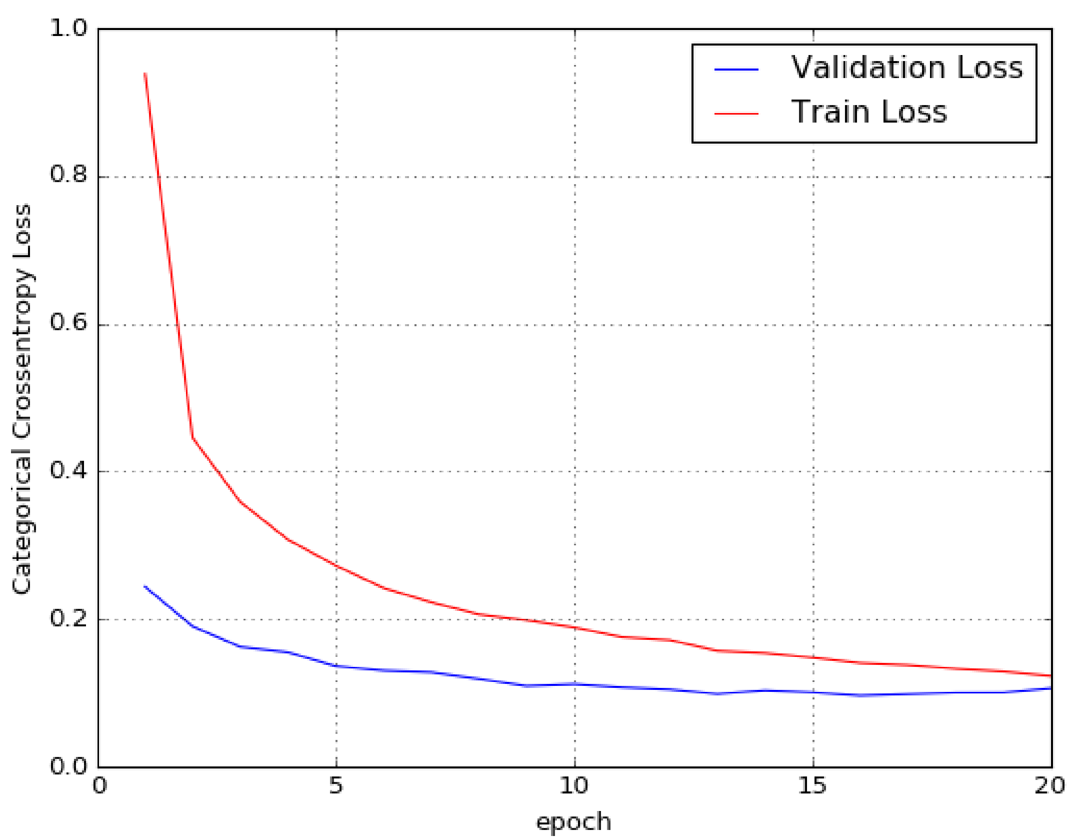
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 11s 175us/step - loss: 0.9381 -
acc: 0.7072 - val_loss: 0.2438 - val_acc: 0.9337
Epoch 2/20
60000/60000 [==============================] - 10s 161us/step - loss: 0.4452 -
acc: 0.8702 - val_loss: 0.1901 - val_acc: 0.9459
Epoch 3/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.3585 -
acc: 0.8967 - val_loss: 0.1622 - val_acc: 0.9531
Epoch 4/20
60000/60000 [==============================] - 10s 174us/step - loss: 0.3074 -
acc: 0.9121 - val_loss: 0.1551 - val_acc: 0.9569
Epoch 5/20
60000/60000 [==============================] - 10s 173us/step - loss: 0.2725 -
acc: 0.9233 - val_loss: 0.1363 - val_acc: 0.9606
Epoch 6/20
60000/60000 [==============================] - 10s 169us/step - loss: 0.2420 -
acc: 0.9331 - val_loss: 0.1302 - val_acc: 0.9638
Epoch 7/20
60000/60000 [==============================] - 11s 181us/step - loss: 0.2228 -
acc: 0.9382 - val_loss: 0.1279 - val_acc: 0.9649
Epoch 8/20
60000/60000 [==============================] - 11s 181us/step - loss: 0.2064 -
acc: 0.9423 - val_loss: 0.1191 - val_acc: 0.9656
Epoch 9/20
60000/60000 [==============================] - 11s 191us/step - loss: 0.1984 -
acc: 0.9451 - val_loss: 0.1094 - val_acc: 0.9698
Epoch 10/20
60000/60000 [==============================] - 12s 193us/step - loss: 0.1887 -
acc: 0.9477 - val_loss: 0.1118 - val_acc: 0.9687
Epoch 11/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.1756 -
acc: 0.9519 - val_loss: 0.1076 - val_acc: 0.9705
Epoch 12/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.1716 -
acc: 0.9527 - val_loss: 0.1048 - val_acc: 0.9720
Epoch 13/20
60000/60000 [==============================] - 11s 184us/step - loss: 0.1570 -
acc: 0.9562 - val_loss: 0.0986 - val_acc: 0.9726
Epoch 14/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.1540 -
acc: 0.9575 - val_loss: 0.1034 - val_acc: 0.9731
Epoch 15/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.1478 - a
cc: 0.9589 - val_loss: 0.1007 - val_acc: 0.9727
Epoch 16/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.1406 - a
cc: 0.9611 - val_loss: 0.0963 - val_acc: 0.9740
Epoch 17/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.1377 - a
cc: 0.9629 - val_loss: 0.0984 - val_acc: 0.9743
Epoch 18/20
```

```
60000/60000 [==============================] - 10s 159us/step - loss: 0.1328 -
acc: 0.9622 - val_loss: 0.1004 - val_acc: 0.9746
Epoch 19/20
60000/60000 [==============================] - 10s 158us/step - loss: 0.1289 -
acc: 0.9637 - val_loss: 0.1005 - val_acc: 0.9753
Epoch 20/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.1230 -
acc: 0.9650 - val_loss: 0.1064 - val_acc: 0.9744
```

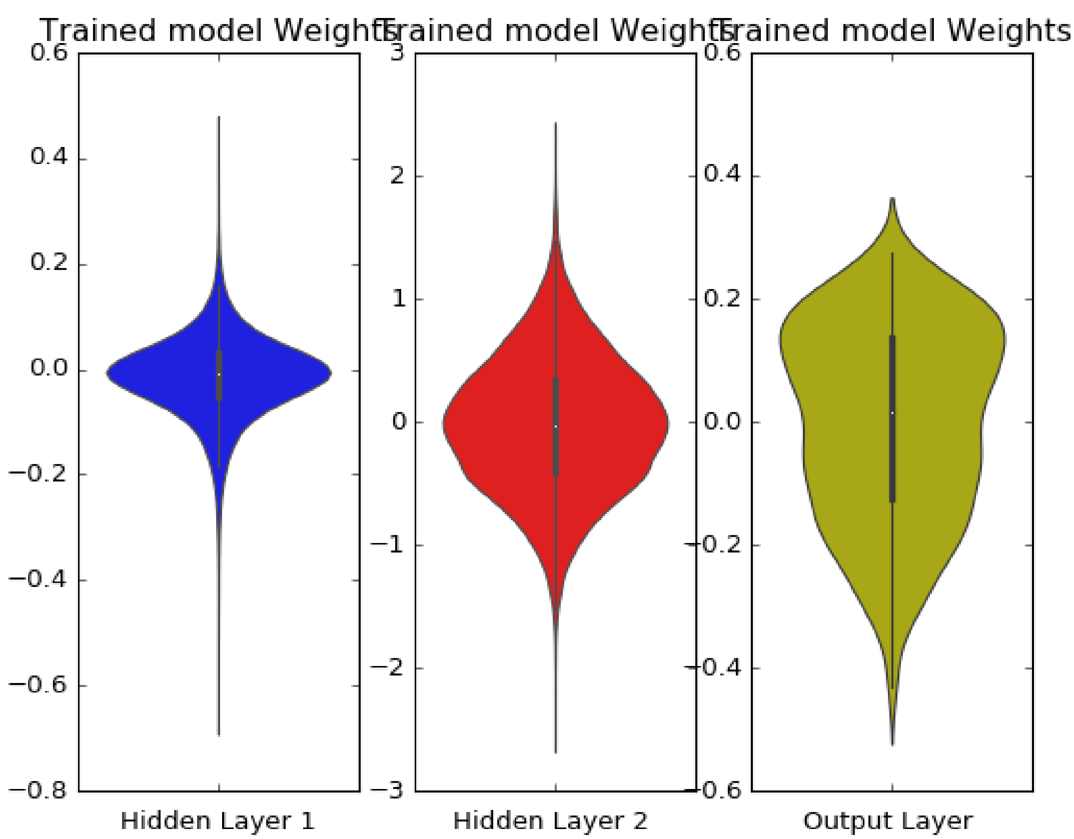
In [37]:

```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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

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

```
Test score: 0.106384716608499
Test accuracy: 0.9744
```



In [38]:
```
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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
```

```
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + BatchNormalization + Dropout + AdamOptimizer with 2 hidden layers

In [34]:

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

from keras.layers import Dropout

model_drop = Sequential()

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

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

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_17 (Dense)             (None, 364)               285740
_________________________________________________________________
dropout_3 (Dropout)          (None, 364)               0
_________________________________________________________________
dense_18 (Dense)             (None, 52)                18980
_________________________________________________________________
dropout_4 (Dropout)          (None, 52)                0
_________________________________________________________________
dense_19 (Dense)             (None, 10)                530
=================================================================
Total params: 305,250
Trainable params: 305,250
Non-trainable params: 0
_________________________________________________________________
```

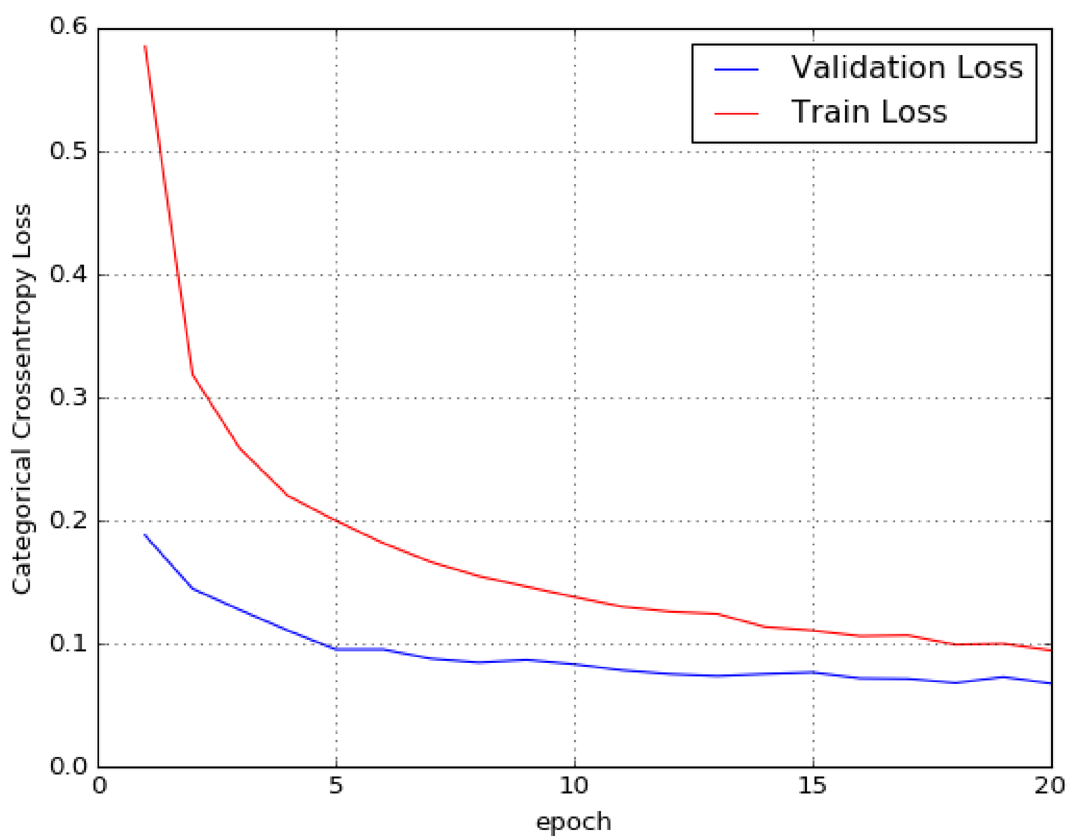
In [31]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 14s 228us/step - loss: 0.5853 -
acc: 0.8241 - val_loss: 0.1880 - val_acc: 0.9449
Epoch 2/20
60000/60000 [==============================] - 11s 188us/step - loss: 0.3184 -
acc: 0.9081 - val_loss: 0.1446 - val_acc: 0.9551
Epoch 3/20
60000/60000 [==============================] - 11s 178us/step - loss: 0.2581 -
acc: 0.9246 - val_loss: 0.1272 - val_acc: 0.9601
Epoch 4/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.2201 -
acc: 0.9359 - val_loss: 0.1106 - val_acc: 0.9650
Epoch 5/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.1999 -
acc: 0.9418 - val_loss: 0.0952 - val_acc: 0.9705
Epoch 6/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.1816 -
acc: 0.9477 - val_loss: 0.0952 - val_acc: 0.9706
Epoch 7/20
60000/60000 [==============================] - 12s 207us/step - loss: 0.1663 -
acc: 0.9518 - val_loss: 0.0877 - val_acc: 0.9737
Epoch 8/20
60000/60000 [==============================] - 12s 207us/step - loss: 0.1548 -
acc: 0.9540 - val_loss: 0.0847 - val_acc: 0.9727
Epoch 9/20
60000/60000 [==============================] - 13s 216us/step - loss: 0.1463 -
acc: 0.9576 - val_loss: 0.0868 - val_acc: 0.9749
Epoch 10/20
60000/60000 [==============================] - 10s 166us/step - loss: 0.1379 -
acc: 0.9588 - val_loss: 0.0833 - val_acc: 0.9750
Epoch 11/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.1301 - a
cc: 0.9624 - val_loss: 0.0784 - val_acc: 0.9766
Epoch 12/20
60000/60000 [==============================] - 10s 169us/step - loss: 0.1260 -
acc: 0.9633 - val_loss: 0.0753 - val_acc: 0.9774
Epoch 13/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.1242 -
acc: 0.9637 - val_loss: 0.0737 - val_acc: 0.9787
Epoch 14/20
60000/60000 [==============================] - 12s 194us/step - loss: 0.1136 -
acc: 0.9663 - val_loss: 0.0753 - val_acc: 0.9788
Epoch 15/20
60000/60000 [==============================] - 12s 195us/step - loss: 0.1106 -
acc: 0.9675 - val_loss: 0.0766 - val_acc: 0.9791
Epoch 16/20
60000/60000 [==============================] - 11s 189us/step - loss: 0.1061 -
acc: 0.9687 - val_loss: 0.0718 - val_acc: 0.9800
Epoch 17/20
60000/60000 [==============================] - 11s 186us/step - loss: 0.1067 -
acc: 0.9676 - val_loss: 0.0714 - val_acc: 0.9795
Epoch 18/20
```

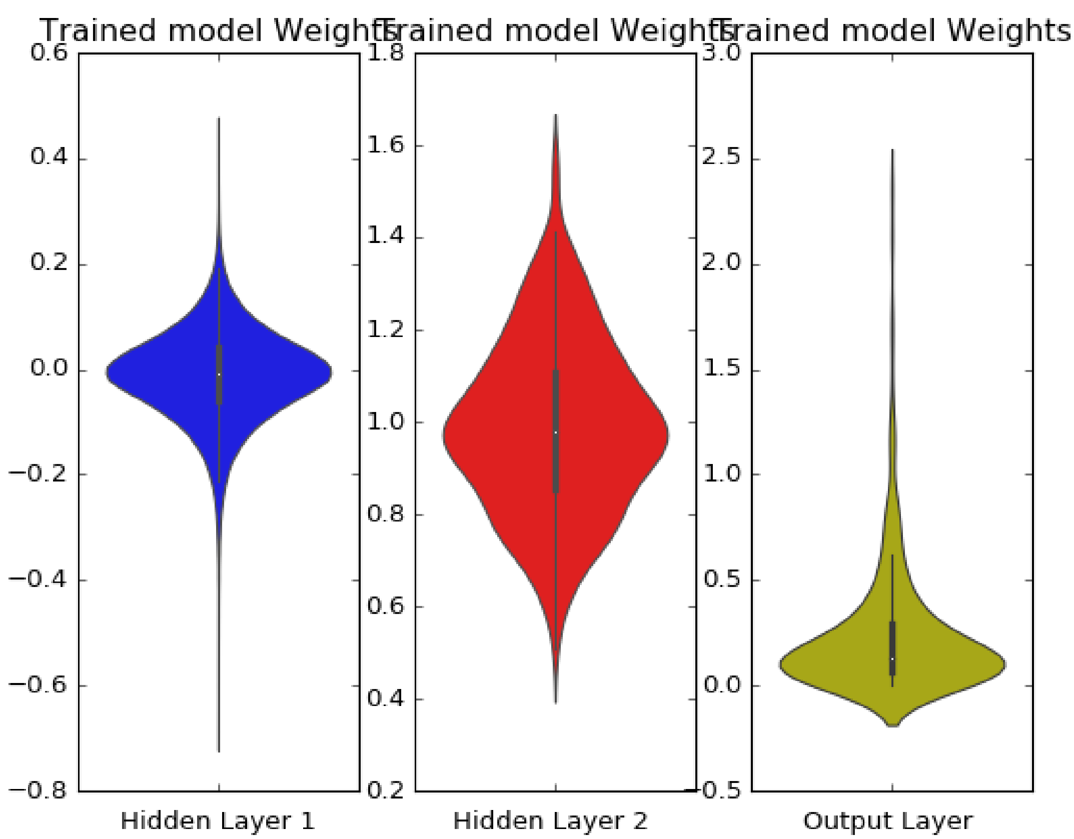
```
60000/60000 [==============================] - 11s 186us/step - loss: 0.0991 -
acc: 0.9708 - val_loss: 0.0681 - val_acc: 0.9800
Epoch 19/20
60000/60000 [==============================] - 11s 190us/step - loss: 0.0999 -
acc: 0.9705 - val_loss: 0.0729 - val_acc: 0.9797
Epoch 20/20
60000/60000 [==============================] - 11s 190us/step - loss: 0.0945 -
acc: 0.9721 - val_loss: 0.0676 - val_acc: 0.9808
```

In [32]:
```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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

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

```
Test score: 0.06761971237978433
Test accuracy: 0.9808
```



In [33]:
```
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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
```

```
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

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

In [55]:
```
model_relu = Sequential()
model_relu.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_ini
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean
model_relu.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

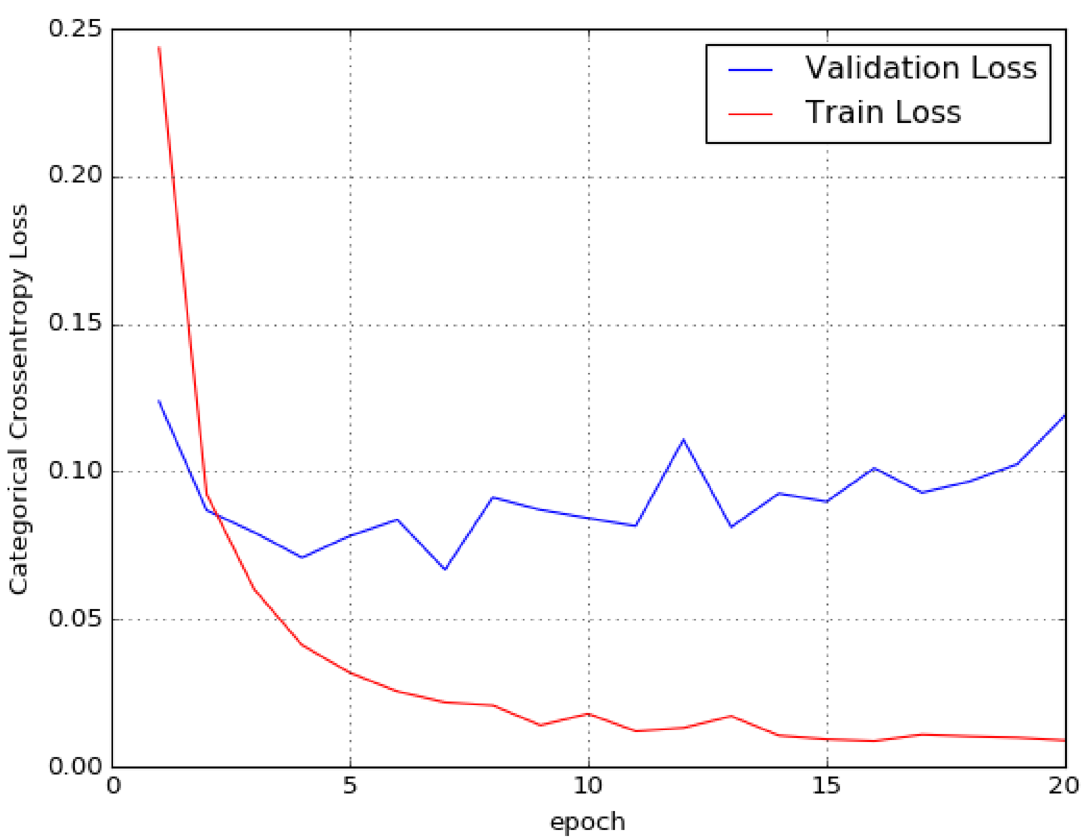
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_35 (Dense)             (None, 364)               285740    
_________________________________________________________________
dense_36 (Dense)             (None, 128)               46720     
_________________________________________________________________
dense_37 (Dense)             (None, 52)                6708      
_________________________________________________________________
dense_38 (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 [==============================] - 14s 236us/step - loss: 0.2435 - 
acc: 0.9263 - val_loss: 0.1237 - val_acc: 0.9618
Epoch 2/20
60000/60000 [==============================] - 11s 183us/step - loss: 0.0923 - 
acc: 0.9721 - val_loss: 0.0868 - val_acc: 0.9727
Epoch 3/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0601 - 
acc: 0.9819 - val_loss: 0.0794 - val_acc: 0.9760
Epoch 4/20
60000/60000 [==============================] - 12s 204us/step - loss: 0.0412 - 
acc: 0.9870 - val_loss: 0.0709 - val_acc: 0.9785
Epoch 5/20
60000/60000 [==============================] - 11s 190us/step - loss: 0.0318 - 
acc: 0.9901 - val_loss: 0.0782 - val_acc: 0.9753
Epoch 6/20
60000/60000 [==============================] - 11s 183us/step - loss: 0.0255 - 
acc: 0.9914 - val_loss: 0.0837 - val_acc: 0.9768
Epoch 7/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0218 - 
acc: 0.9930 - val_loss: 0.0668 - val_acc: 0.9804
Epoch 8/20
60000/60000 [==============================] - 11s 188us/step - loss: 0.0208 - 
acc: 0.9930 - val_loss: 0.0912 - val_acc: 0.9766
Epoch 9/20
60000/60000 [==============================] - 11s 184us/step - loss: 0.0141 - 
acc: 0.9955 - val_loss: 0.0870 - val_acc: 0.9775
```

```
Epoch 10/20
60000/60000 [==============================] - 12s 201us/step - loss: 0.0179 -
acc: 0.9938 - val_loss: 0.0842 - val_acc: 0.9783
Epoch 11/20
60000/60000 [==============================] - 12s 196us/step - loss: 0.0120 -
acc: 0.9962 - val_loss: 0.0815 - val_acc: 0.9794
Epoch 12/20
60000/60000 [==============================] - 11s 184us/step - loss: 0.0132 -
acc: 0.9958 - val_loss: 0.1108 - val_acc: 0.9761
Epoch 13/20
60000/60000 [==============================] - 11s 183us/step - loss: 0.0171 -
acc: 0.9945 - val_loss: 0.0812 - val_acc: 0.9820
Epoch 14/20
60000/60000 [==============================] - 11s 191us/step - loss: 0.0106 -
acc: 0.9967 - val_loss: 0.0926 - val_acc: 0.9809
Epoch 15/20
60000/60000 [==============================] - 11s 190us/step - loss: 0.0094 -
acc: 0.9969 - val_loss: 0.0898 - val_acc: 0.9808
Epoch 16/20
60000/60000 [==============================] - 11s 189us/step - loss: 0.0088 -
acc: 0.9970 - val_loss: 0.1011 - val_acc: 0.9790
Epoch 17/20
60000/60000 [==============================] - 11s 188us/step - loss: 0.0109 -
acc: 0.9965 - val_loss: 0.0928 - val_acc: 0.9819
Epoch 18/20
60000/60000 [==============================] - 11s 187us/step - loss: 0.0103 -
acc: 0.9963 - val_loss: 0.0966 - val_acc: 0.9814
Epoch 19/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.0099 -
acc: 0.9966 - val_loss: 0.1025 - val_acc: 0.9804
Epoch 20/20
60000/60000 [==============================] - 11s 191us/step - loss: 0.0090 -
acc: 0.9972 - val_loss: 0.1190 - val_acc: 0.9787
```

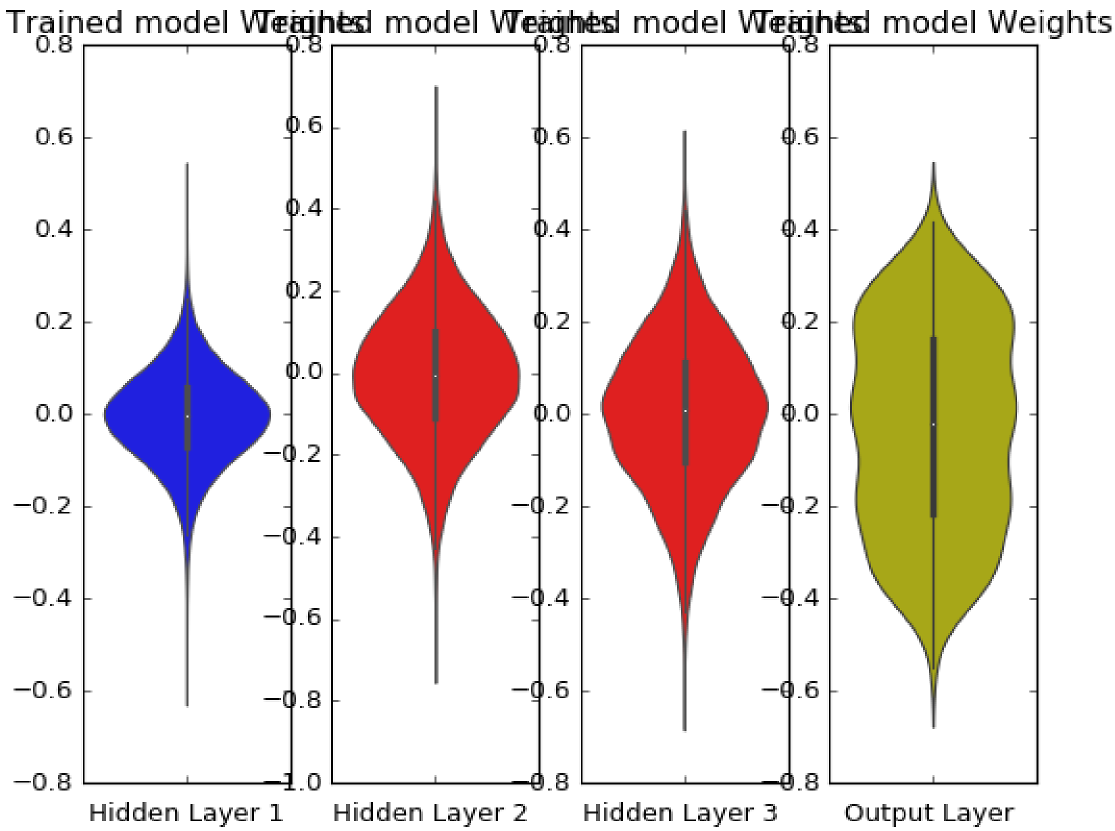
In [56]:

```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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


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

```
Test score: 0.11902276019332958
Test accuracy: 0.9787
```



In [60]:
```
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='r')
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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + Batch-Norm on 3 hidden Layers + AdamOptimizer

In [61]:
```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this conditio
# 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_in
model_batch.add(BatchNormalization())

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

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

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_39 (Dense)             (None, 364)               285740
_________________________________________________________________
batch_normalization_10 (Batc (None, 364)               1456
_________________________________________________________________
dense_40 (Dense)             (None, 128)               46720
_________________________________________________________________
batch_normalization_11 (Batc (None, 128)               512
_________________________________________________________________
dense_41 (Dense)             (None, 52)                6708
_________________________________________________________________
batch_normalization_12 (Batc (None, 52)                208
_________________________________________________________________
dense_42 (Dense)             (None, 10)                530
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

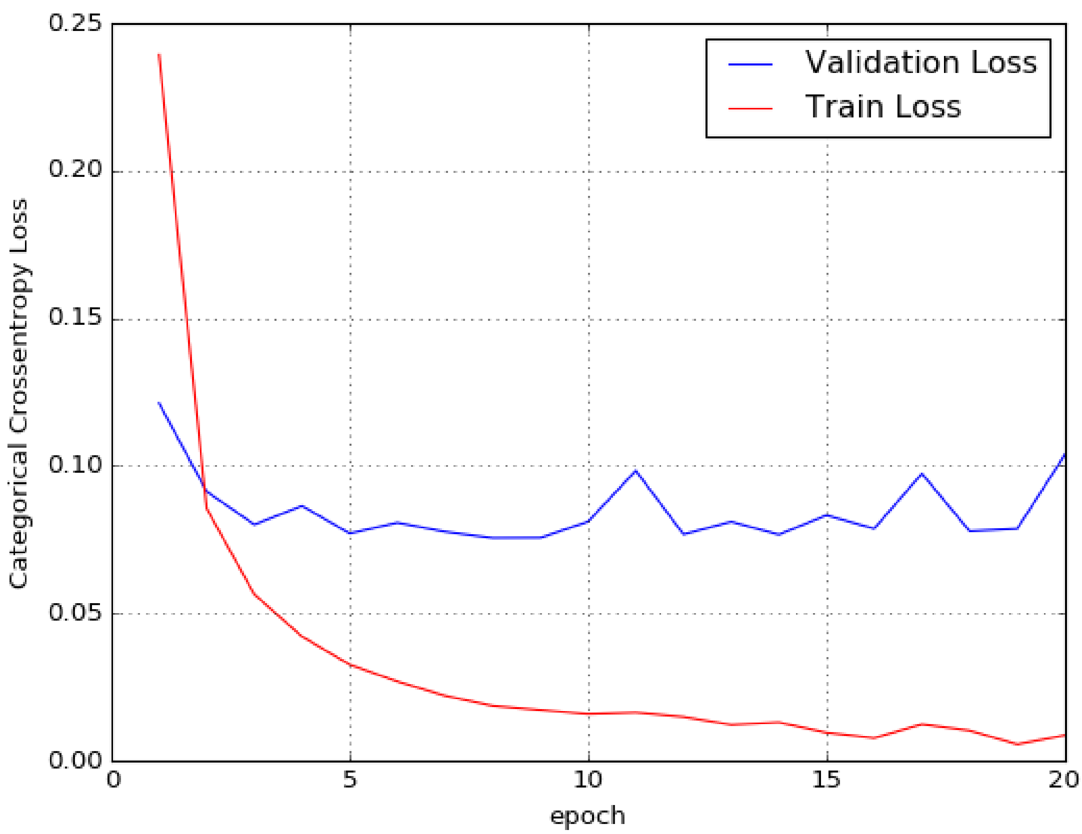
In [62]:
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 14s 227us/step - loss: 0.2393 -
acc: 0.9323 - val_loss: 0.1212 - val_acc: 0.9651
Epoch 2/20
60000/60000 [==============================] - 12s 193us/step - loss: 0.0855 -
acc: 0.9751 - val_loss: 0.0913 - val_acc: 0.9732
Epoch 3/20
60000/60000 [==============================] - 12s 201us/step - loss: 0.0564 -
acc: 0.9828 - val_loss: 0.0801 - val_acc: 0.9749
Epoch 4/20
60000/60000 [==============================] - 12s 201us/step - loss: 0.0421 -
acc: 0.9868 - val_loss: 0.0864 - val_acc: 0.9741
Epoch 5/20
60000/60000 [==============================] - 13s 222us/step - loss: 0.0326 -
acc: 0.9897 - val_loss: 0.0771 - val_acc: 0.9774
Epoch 6/20
60000/60000 [==============================] - 13s 223us/step - loss: 0.0269 -
acc: 0.9915 - val_loss: 0.0807 - val_acc: 0.9780
Epoch 7/20
60000/60000 [==============================] - 14s 233us/step - loss: 0.0220 -
acc: 0.9928 - val_loss: 0.0776 - val_acc: 0.9773
Epoch 8/20
60000/60000 [==============================] - 14s 229us/step - loss: 0.0186 -
acc: 0.9938 - val_loss: 0.0755 - val_acc: 0.9806
Epoch 9/20
60000/60000 [==============================] - 12s 205us/step - loss: 0.0172 -
acc: 0.9946 - val_loss: 0.0756 - val_acc: 0.9787
Epoch 10/20
60000/60000 [==============================] - 13s 211us/step - loss: 0.0158 -
acc: 0.9948 - val_loss: 0.0811 - val_acc: 0.9774
Epoch 11/20
60000/60000 [==============================] - 12s 206us/step - loss: 0.0164 -
acc: 0.9942 - val_loss: 0.0984 - val_acc: 0.9764
Epoch 12/20
60000/60000 [==============================] - 12s 200us/step - loss: 0.0149 -
acc: 0.9949 - val_loss: 0.0768 - val_acc: 0.9802
Epoch 13/20
60000/60000 [==============================] - 13s 214us/step - loss: 0.0122 -
acc: 0.9959 - val_loss: 0.0810 - val_acc: 0.9809
Epoch 14/20
60000/60000 [==============================] - 13s 221us/step - loss: 0.0130 -
acc: 0.9958 - val_loss: 0.0767 - val_acc: 0.9787
Epoch 15/20
60000/60000 [==============================] - 13s 210us/step - loss: 0.0095 -
acc: 0.9968 - val_loss: 0.0832 - val_acc: 0.9797
Epoch 16/20
60000/60000 [==============================] - 12s 201us/step - loss: 0.0077 -
acc: 0.9975 - val_loss: 0.0787 - val_acc: 0.9794
Epoch 17/20
60000/60000 [==============================] - 12s 194us/step - loss: 0.0123 -
acc: 0.9958 - val_loss: 0.0974 - val_acc: 0.9765
Epoch 18/20
```

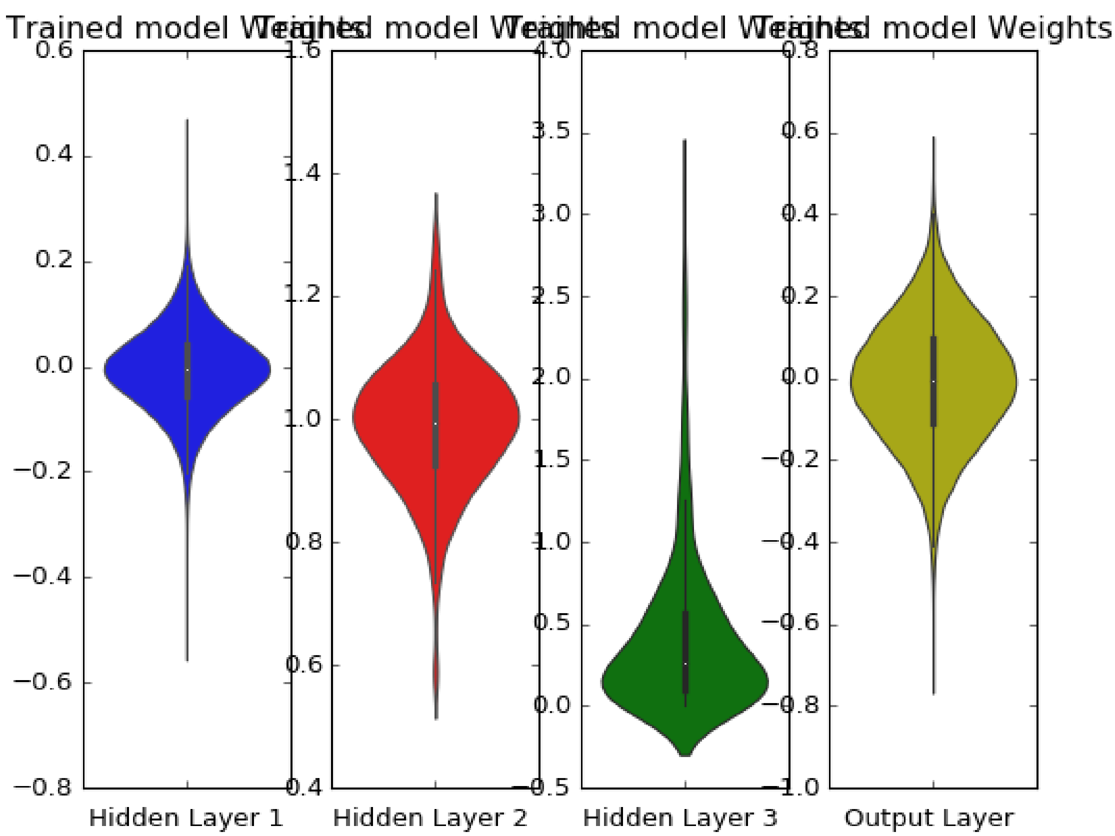
```
60000/60000 [==============================] - 12s 194us/step - loss: 0.0102 -
acc: 0.9966 - val_loss: 0.0778 - val_acc: 0.9806
Epoch 19/20
60000/60000 [==============================] - 12s 204us/step - loss: 0.0056 -
acc: 0.9982 - val_loss: 0.0787 - val_acc: 0.9812
Epoch 20/20
60000/60000 [==============================] - 12s 193us/step - loss: 0.0086 -
acc: 0.9972 - val_loss: 0.1038 - val_acc: 0.9759
```

In [63]:
```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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

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

```
Test score: 0.10380259412173763
Test accuracy: 0.9759
```



In [64]:
```
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()
```



```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + Dropout + AdamOptimizer with 3 hidden layers

In [65]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormaliza

from keras.layers import Dropout

model_drop = Sequential()

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

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

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

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_43 (Dense)             (None, 364)               285740
_________________________________________________________________
dropout_10 (Dropout)         (None, 364)               0
_________________________________________________________________
dense_44 (Dense)             (None, 128)               46720
_________________________________________________________________
dropout_11 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_45 (Dense)             (None, 52)                6708
_________________________________________________________________
dropout_12 (Dropout)         (None, 52)                0
_________________________________________________________________
dense_46 (Dense)             (None, 10)                530
=================================================================
Total params: 339,698
Trainable params: 339,698
Non-trainable params: 0
_________________________________________________________________
```

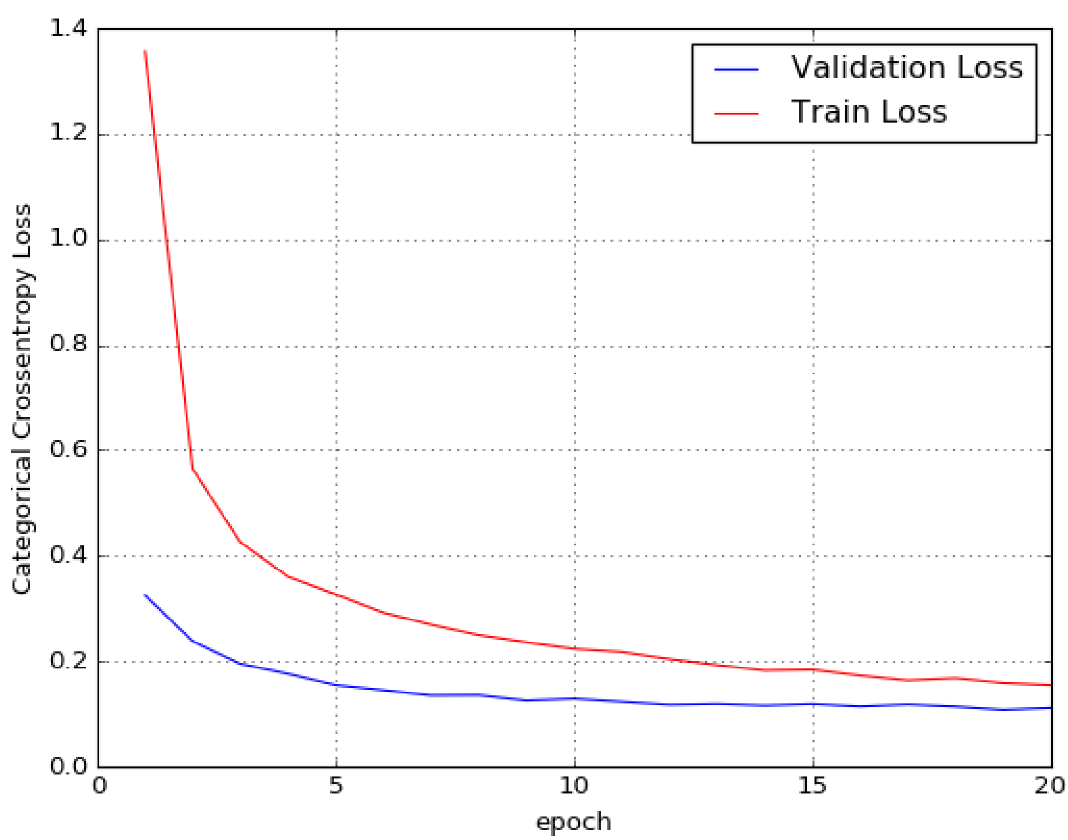
In [66]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 12s 205us/step - loss: 1.3565 -
acc: 0.5857 - val_loss: 0.3252 - val_acc: 0.9171
Epoch 2/20
60000/60000 [==============================] - 9s 155us/step - loss: 0.5646 - a
cc: 0.8396 - val_loss: 0.2378 - val_acc: 0.9411
Epoch 3/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.4256 - a
cc: 0.8865 - val_loss: 0.1949 - val_acc: 0.9506
Epoch 4/20
60000/60000 [==============================] - 10s 162us/step - loss: 0.3604 -
acc: 0.9068 - val_loss: 0.1762 - val_acc: 0.9568
Epoch 5/20
60000/60000 [==============================] - 10s 167us/step - loss: 0.3266 -
acc: 0.9173 - val_loss: 0.1547 - val_acc: 0.9612
Epoch 6/20
60000/60000 [==============================] - 9s 155us/step - loss: 0.2920 - a
cc: 0.9252 - val_loss: 0.1450 - val_acc: 0.9621
Epoch 7/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.2694 - a
cc: 0.9332 - val_loss: 0.1352 - val_acc: 0.9662
Epoch 8/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.2495 -
acc: 0.9369 - val_loss: 0.1357 - val_acc: 0.9651
Epoch 9/20
60000/60000 [==============================] - 12s 206us/step - loss: 0.2358 -
acc: 0.9419 - val_loss: 0.1256 - val_acc: 0.9682
Epoch 10/20
60000/60000 [==============================] - 10s 160us/step - loss: 0.2235 -
acc: 0.9438 - val_loss: 0.1291 - val_acc: 0.9685
Epoch 11/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.2175 -
acc: 0.9446 - val_loss: 0.1233 - val_acc: 0.9691
Epoch 12/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.2040 - a
cc: 0.9483 - val_loss: 0.1175 - val_acc: 0.9708
Epoch 13/20
60000/60000 [==============================] - 10s 166us/step - loss: 0.1926 -
acc: 0.9515 - val_loss: 0.1191 - val_acc: 0.9714
Epoch 14/20
60000/60000 [==============================] - 10s 170us/step - loss: 0.1826 -
acc: 0.9534 - val_loss: 0.1161 - val_acc: 0.9722
Epoch 15/20
60000/60000 [==============================] - 11s 188us/step - loss: 0.1841 -
acc: 0.9549 - val_loss: 0.1190 - val_acc: 0.9714
Epoch 16/20
60000/60000 [==============================] - 11s 188us/step - loss: 0.1730 -
acc: 0.9566 - val_loss: 0.1142 - val_acc: 0.9737
Epoch 17/20
60000/60000 [==============================] - 11s 190us/step - loss: 0.1637 -
acc: 0.9591 - val_loss: 0.1182 - val_acc: 0.9732
Epoch 18/20
```

```
60000/60000 [==============================] - 11s 184us/step - loss: 0.1679 -
acc: 0.9587 - val_loss: 0.1139 - val_acc: 0.9738
Epoch 19/20
60000/60000 [==============================] - 12s 192us/step - loss: 0.1586 -
acc: 0.9605 - val_loss: 0.1078 - val_acc: 0.9762
Epoch 20/20
60000/60000 [==============================] - 11s 188us/step - loss: 0.1546 -
acc: 0.9600 - val_loss: 0.1115 - val_acc: 0.9738
```

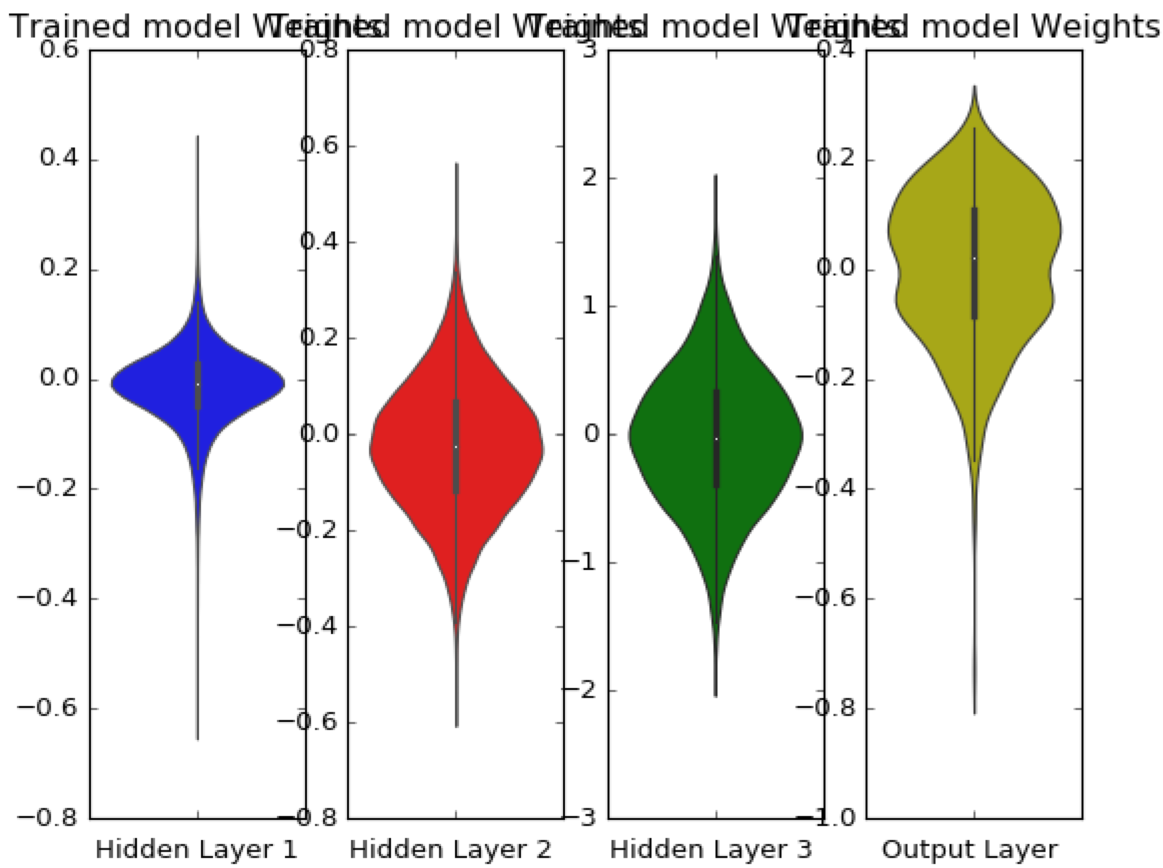
In [67]:
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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

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

```
Test score: 0.1115148579780769
Test accuracy: 0.9738
```



In [68]:
```
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()
```

Figure 1




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + BatchNormalization + Dropout + AdamOptimizer with 3 hidden layers

In [69]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormaliza

from keras.layers import Dropout

model_drop = Sequential()

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

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


model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=
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_47 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_13 (Batc (None, 364)               1456      
_________________________________________________________________
dropout_13 (Dropout)         (None, 364)               0         
_________________________________________________________________
dense_48 (Dense)             (None, 128)               46720     
_________________________________________________________________
batch_normalization_14 (Batc (None, 128)               512       
_________________________________________________________________
dropout_14 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_49 (Dense)             (None, 52)                6708      
_________________________________________________________________
batch_normalization_15 (Batc (None, 52)                208       
_________________________________________________________________
dropout_15 (Dropout)         (None, 52)                0         
_________________________________________________________________
dense_50 (Dense)             (None, 10)                530       
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

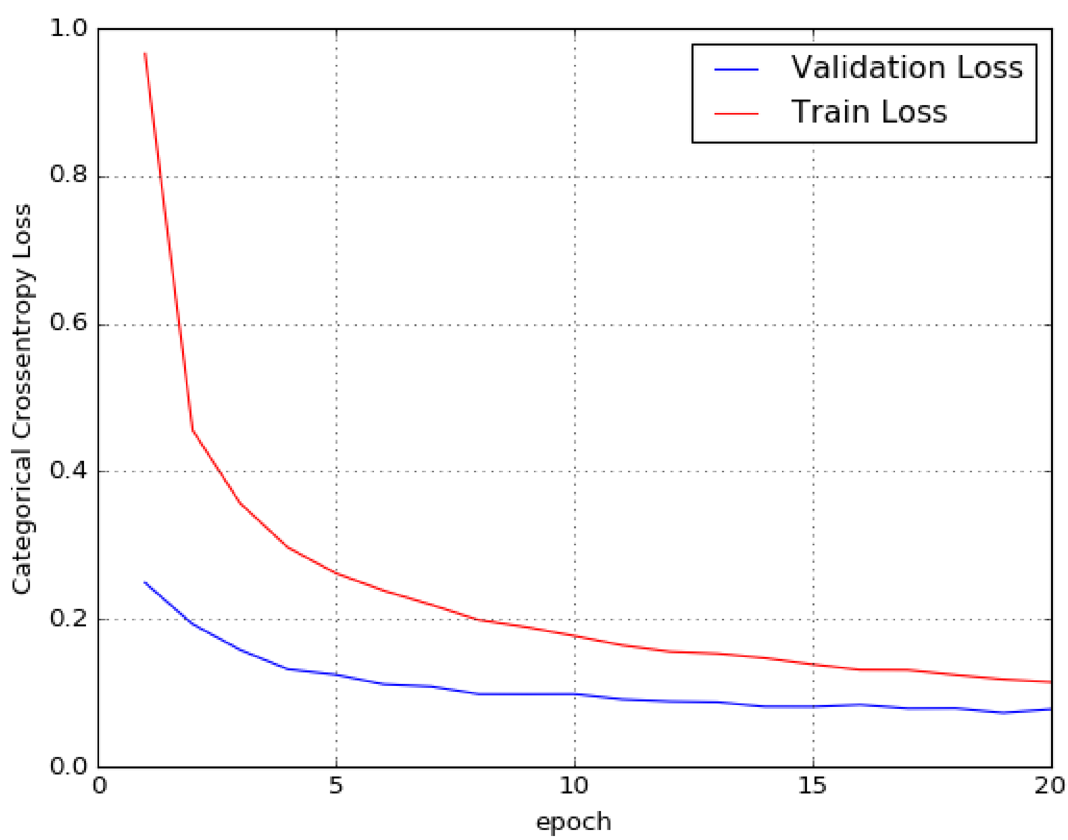
In [70]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 15s 244us/step - loss: 0.9654 -
acc: 0.6985 - val_loss: 0.2492 - val_acc: 0.9254
Epoch 2/20
60000/60000 [==============================] - 12s 196us/step - loss: 0.4556 -
acc: 0.8676 - val_loss: 0.1930 - val_acc: 0.9420
Epoch 3/20
60000/60000 [==============================] - 12s 199us/step - loss: 0.3565 -
acc: 0.8991 - val_loss: 0.1581 - val_acc: 0.9530
Epoch 4/20
60000/60000 [==============================] - 11s 181us/step - loss: 0.2967 -
acc: 0.9176 - val_loss: 0.1319 - val_acc: 0.9612
Epoch 5/20
60000/60000 [==============================] - 14s 228us/step - loss: 0.2620 -
acc: 0.9276 - val_loss: 0.1246 - val_acc: 0.9618
Epoch 6/20
60000/60000 [==============================] - 13s 220us/step - loss: 0.2385 -
acc: 0.9355 - val_loss: 0.1116 - val_acc: 0.9678
Epoch 7/20
60000/60000 [==============================] - 13s 218us/step - loss: 0.2194 -
acc: 0.9394 - val_loss: 0.1086 - val_acc: 0.9696
Epoch 8/20
60000/60000 [==============================] - 12s 207us/step - loss: 0.1987 -
acc: 0.9453 - val_loss: 0.0983 - val_acc: 0.9711
Epoch 9/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.1889 -
acc: 0.9483 - val_loss: 0.0982 - val_acc: 0.9726
Epoch 10/20
60000/60000 [==============================] - 11s 178us/step - loss: 0.1772 -
acc: 0.9508 - val_loss: 0.0983 - val_acc: 0.9715
Epoch 11/20
60000/60000 [==============================] - 11s 187us/step - loss: 0.1645 -
acc: 0.9548 - val_loss: 0.0912 - val_acc: 0.9749
Epoch 12/20
60000/60000 [==============================] - 13s 217us/step - loss: 0.1559 -
acc: 0.9561 - val_loss: 0.0884 - val_acc: 0.9749
Epoch 13/20
60000/60000 [==============================] - 12s 207us/step - loss: 0.1531 -
acc: 0.9581 - val_loss: 0.0874 - val_acc: 0.9769
Epoch 14/20
60000/60000 [==============================] - 13s 217us/step - loss: 0.1472 -
acc: 0.9598 - val_loss: 0.0812 - val_acc: 0.9767
Epoch 15/20
60000/60000 [==============================] - 13s 216us/step - loss: 0.1385 -
acc: 0.9629 - val_loss: 0.0811 - val_acc: 0.9777
Epoch 16/20
60000/60000 [==============================] - 13s 211us/step - loss: 0.1311 -
acc: 0.9638 - val_loss: 0.0839 - val_acc: 0.9772
Epoch 17/20
60000/60000 [==============================] - 13s 213us/step - loss: 0.1308 -
acc: 0.9638 - val_loss: 0.0787 - val_acc: 0.9774
Epoch 18/20
```

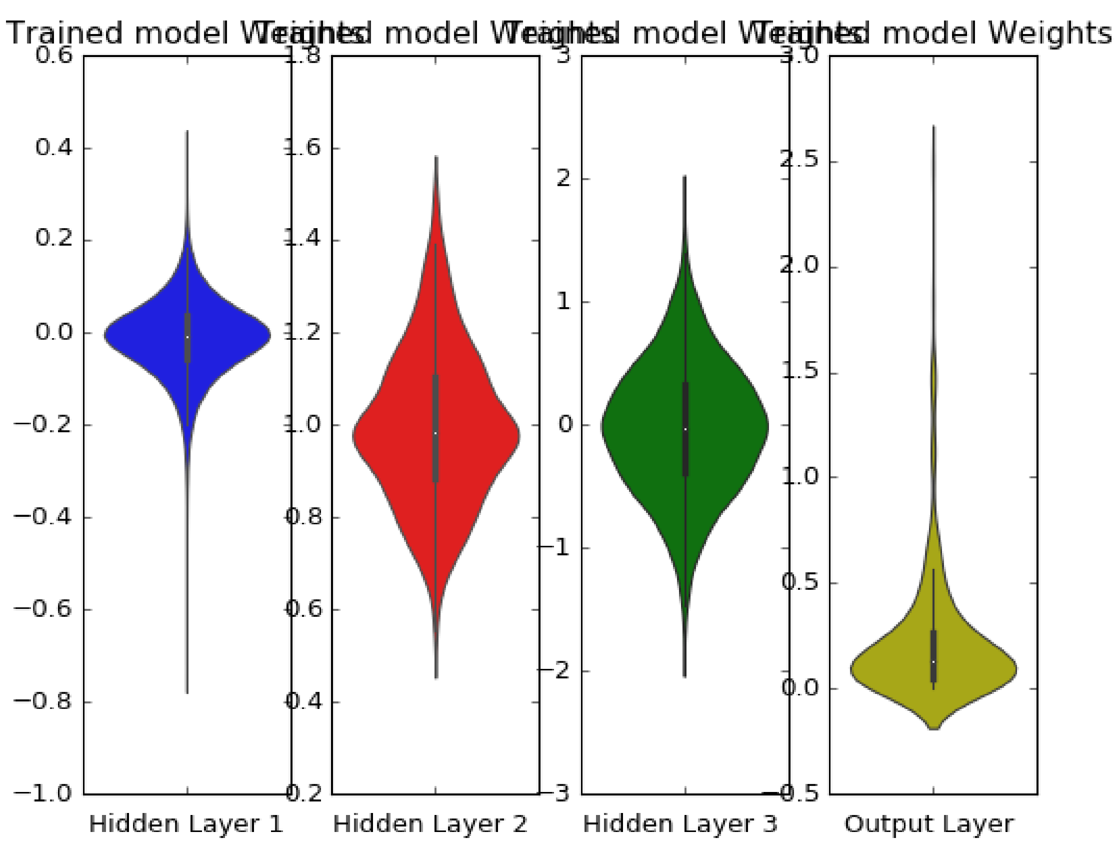
```
60000/60000 [==============================] - 12s 207us/step - loss: 0.1242 -
acc: 0.9658 - val_loss: 0.0790 - val_acc: 0.9783
Epoch 19/20
60000/60000 [==============================] - 13s 214us/step - loss: 0.1183 -
acc: 0.9678 - val_loss: 0.0730 - val_acc: 0.9809
Epoch 20/20
60000/60000 [==============================] - 12s 202us/step - loss: 0.1146 -
acc: 0.9688 - val_loss: 0.0778 - val_acc: 0.9784
```

In [71]:
```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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

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

```
Test score: 0.07784969780180837
Test accuracy: 0.9784
```



In [72]:
```
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()
```



```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

In [ ]:

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

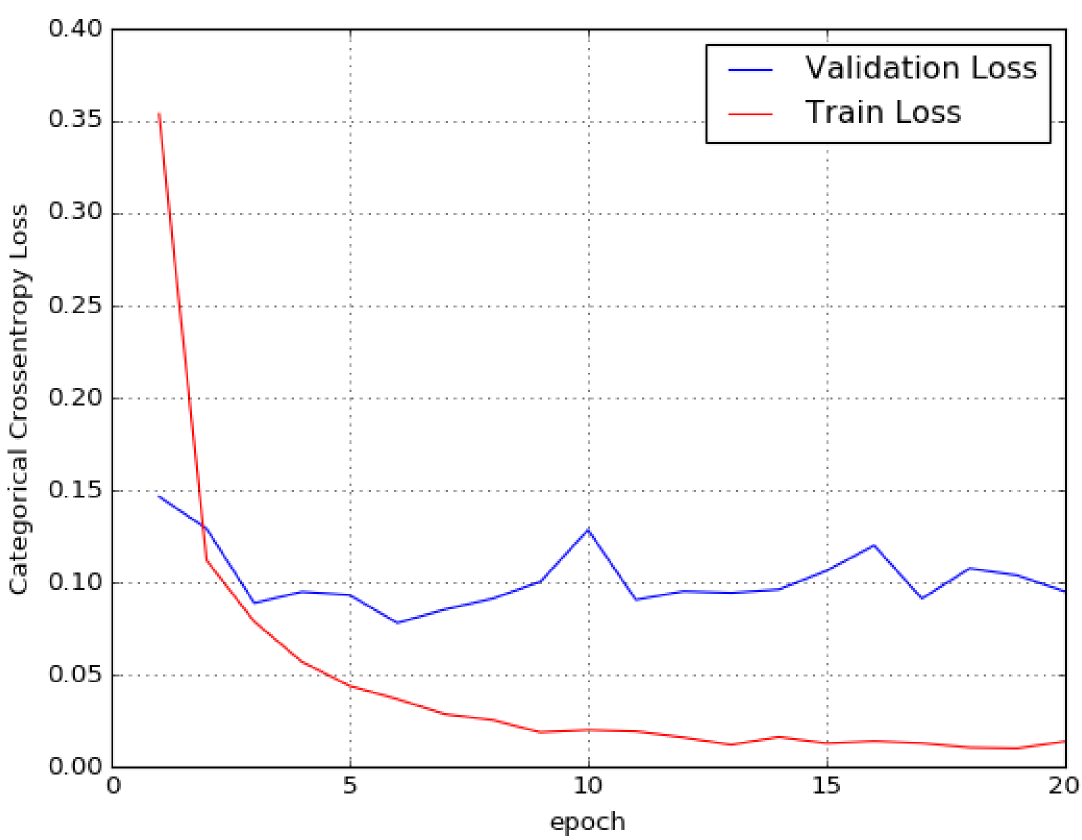
In [73]:
```
model_relu = Sequential()
model_relu.add(Dense(256, activation='relu', input_shape=(input_dim,), kernel_ini
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean
model_relu.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=
model_relu.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=
model_relu.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

model_relu.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_51 (Dense)             (None, 256)               200960    
_________________________________________________________________
dense_52 (Dense)             (None, 128)               32896     
_________________________________________________________________
dense_53 (Dense)             (None, 64)                8256      
_________________________________________________________________
dense_54 (Dense)             (None, 32)                2080      
_________________________________________________________________
dense_55 (Dense)             (None, 16)                528       
_________________________________________________________________
dense_56 (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 [==============================] - 10s 161us/step - loss: 0.3539 - 
acc: 0.8935 - val_loss: 0.1465 - val_acc: 0.9570
Epoch 2/20
60000/60000 [==============================] - 7s 119us/step - loss: 0.1118 - a
cc: 0.9668 - val_loss: 0.1290 - val_acc: 0.9642
Epoch 3/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.0786 - a
cc: 0.9766 - val_loss: 0.0887 - val_acc: 0.9746
Epoch 4/20
60000/60000 [==============================] - 7s 108us/step - loss: 0.0568 - a
cc: 0.9831 - val_loss: 0.0948 - val_acc: 0.9729
Epoch 5/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0438 - a
cc: 0.9861 - val_loss: 0.0930 - val_acc: 0.9731
Epoch 6/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0366 - a
cc: 0.9878 - val_loss: 0.0779 - val_acc: 0.9767
Epoch 7/20
60000/60000 [==============================] - 7s 119us/step - loss: 0.0285 - a
cc: 0.9909 - val_loss: 0.0853 - val_acc: 0.9760
```

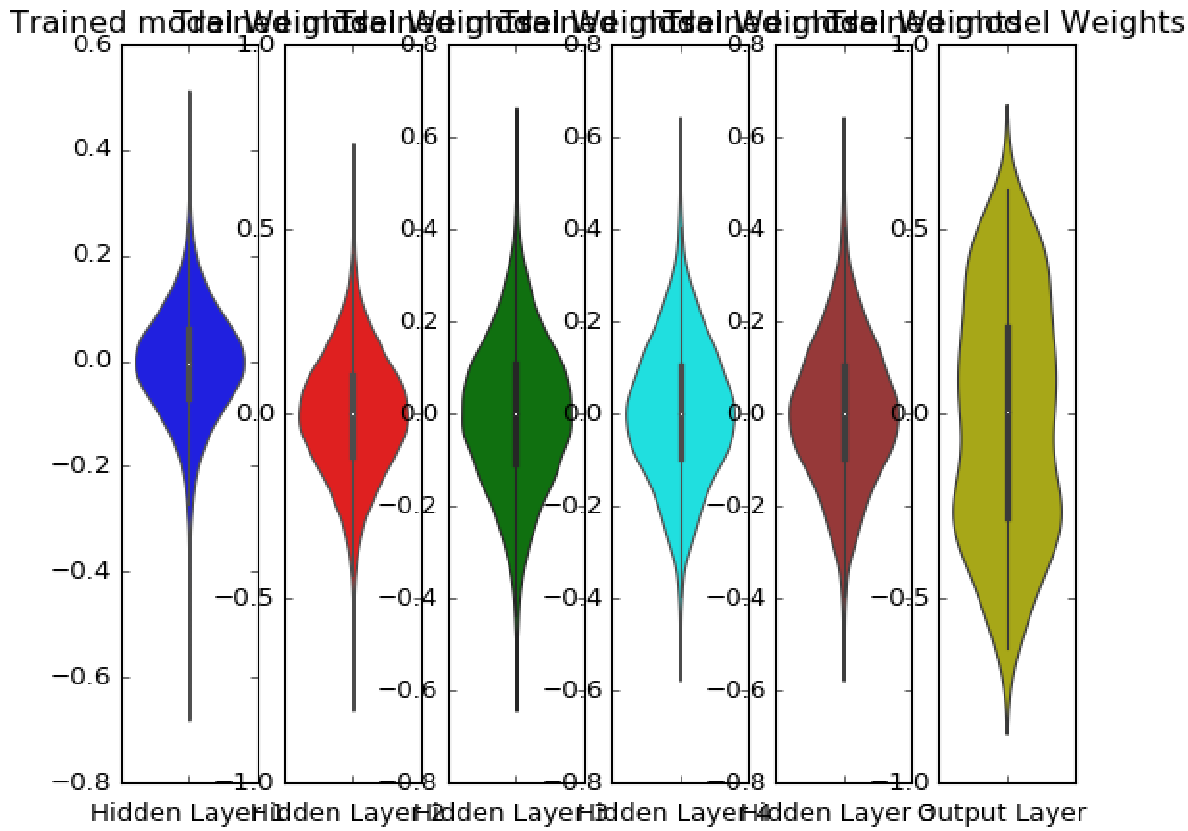
```
Epoch 8/20
60000/60000 [==============================] - 6s 108us/step - loss: 0.0253 - a
cc: 0.9914 - val_loss: 0.0911 - val_acc: 0.9763
Epoch 9/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.0187 - a
cc: 0.9937 - val_loss: 0.1003 - val_acc: 0.9752
Epoch 10/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0200 - ac
c: 0.9936 - val_loss: 0.1284 - val_acc: 0.9674
Epoch 11/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.0191 - a
cc: 0.9936 - val_loss: 0.0905 - val_acc: 0.9775
Epoch 12/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0159 - ac
c: 0.9945 - val_loss: 0.0951 - val_acc: 0.9773
Epoch 13/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0119 - ac
c: 0.9962 - val_loss: 0.0942 - val_acc: 0.9801
Epoch 14/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0162 - ac
c: 0.9947 - val_loss: 0.0961 - val_acc: 0.9775
Epoch 15/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0127 - ac
c: 0.9960 - val_loss: 0.1063 - val_acc: 0.9775
Epoch 16/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.0139 - a
cc: 0.9956 - val_loss: 0.1199 - val_acc: 0.9762
Epoch 17/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.0128 - a
cc: 0.9960 - val_loss: 0.0911 - val_acc: 0.9807
Epoch 18/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0104 - a
cc: 0.9970 - val_loss: 0.1075 - val_acc: 0.9775
Epoch 19/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0100 - a
cc: 0.9970 - val_loss: 0.1037 - val_acc: 0.9791
Epoch 20/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0137 - ac
c: 0.9957 - val_loss: 0.0950 - val_acc: 0.9809
```

In [74]:
```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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


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

```
Test score: 0.09499137327706776
Test accuracy: 0.9809
```



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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + Batch-Norm on 5 hidden Layers + AdamOptimizer

In [79]:
```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this conditio
# 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_in
model_batch.add(BatchNormalization())

model_batch.add(Dense(132, activation='relu', kernel_initializer=RandomNormal(mea
model_batch.add(BatchNormalization())

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

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

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

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_57 (Dense)             (None, 256)               200960    
_________________________________________________________________
batch_normalization_16 (Batc (None, 256)               1024      
_________________________________________________________________
dense_58 (Dense)             (None, 132)               33924     
_________________________________________________________________
batch_normalization_17 (Batc (None, 132)               528       
_________________________________________________________________
dense_59 (Dense)             (None, 64)                8512      
_________________________________________________________________
batch_normalization_18 (Batc (None, 64)                256       
_________________________________________________________________
dense_60 (Dense)             (None, 32)                2080      
_________________________________________________________________
batch_normalization_19 (Batc (None, 32)                128       
_________________________________________________________________
dense_61 (Dense)             (None, 16)                528       
_________________________________________________________________
batch_normalization_20 (Batc (None, 16)                64        
_________________________________________________________________
dense_62 (Dense)             (None, 10)                170       
```

```
=================================================================
Total params: 248,174
Trainable params: 247,174
Non-trainable params: 1,000
_________________________________________________________________
```

In [80]:
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 13s 215us/step - loss: 0.3855 -
acc: 0.9065 - val_loss: 0.1549 - val_acc: 0.9567
Epoch 2/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.1210 - a
cc: 0.9662 - val_loss: 0.1150 - val_acc: 0.9672
Epoch 3/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.0813 - a
cc: 0.9761 - val_loss: 0.1012 - val_acc: 0.9709
Epoch 4/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0631 - a
cc: 0.9806 - val_loss: 0.0808 - val_acc: 0.9769
Epoch 5/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0512 - a
cc: 0.9840 - val_loss: 0.1057 - val_acc: 0.9674
Epoch 6/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.0426 - a
cc: 0.9870 - val_loss: 0.0886 - val_acc: 0.9744
Epoch 7/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.0331 - a
cc: 0.9896 - val_loss: 0.0946 - val_acc: 0.9729
Epoch 8/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.0330 - a
cc: 0.9895 - val_loss: 0.0868 - val_acc: 0.9762
Epoch 9/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.0313 - a
cc: 0.9899 - val_loss: 0.0887 - val_acc: 0.9759
Epoch 10/20
60000/60000 [==============================] - 10s 160us/step - loss: 0.0246 -
acc: 0.9916 - val_loss: 0.0869 - val_acc: 0.9760
Epoch 11/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0260 - a
cc: 0.9913 - val_loss: 0.0831 - val_acc: 0.9779
Epoch 12/20
60000/60000 [==============================] - 8s 135us/step - loss: 0.0216 - a
cc: 0.9928 - val_loss: 0.0854 - val_acc: 0.9783
Epoch 13/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.0201 - a
cc: 0.9937 - val_loss: 0.0865 - val_acc: 0.9784
Epoch 14/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.0176 - a
cc: 0.9945 - val_loss: 0.0784 - val_acc: 0.9794
Epoch 15/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.0165 - a
cc: 0.9946 - val_loss: 0.0824 - val_acc: 0.9781
Epoch 16/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.0150 - a
cc: 0.9950 - val_loss: 0.0869 - val_acc: 0.9804
Epoch 17/20
60000/60000 [==============================] - 10s 166us/step - loss: 0.0164 -
acc: 0.9944 - val_loss: 0.0667 - val_acc: 0.9820
Epoch 18/20
```

```
60000/60000 [==============================] - 9s 158us/step - loss: 0.0136 - a
cc: 0.9954 - val_loss: 0.0785 - val_acc: 0.9811
Epoch 19/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.0158 - a
cc: 0.9947 - val_loss: 0.0868 - val_acc: 0.9791
Epoch 20/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.0139 - a
cc: 0.9956 - val_loss: 0.0819 - val_acc: 0.9814
```

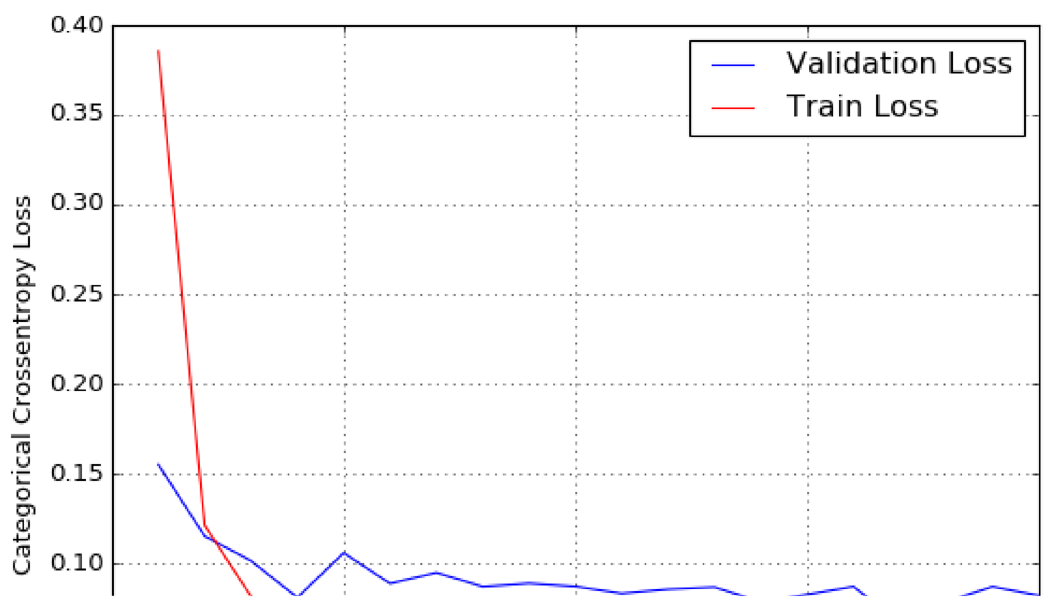
In [84]:

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

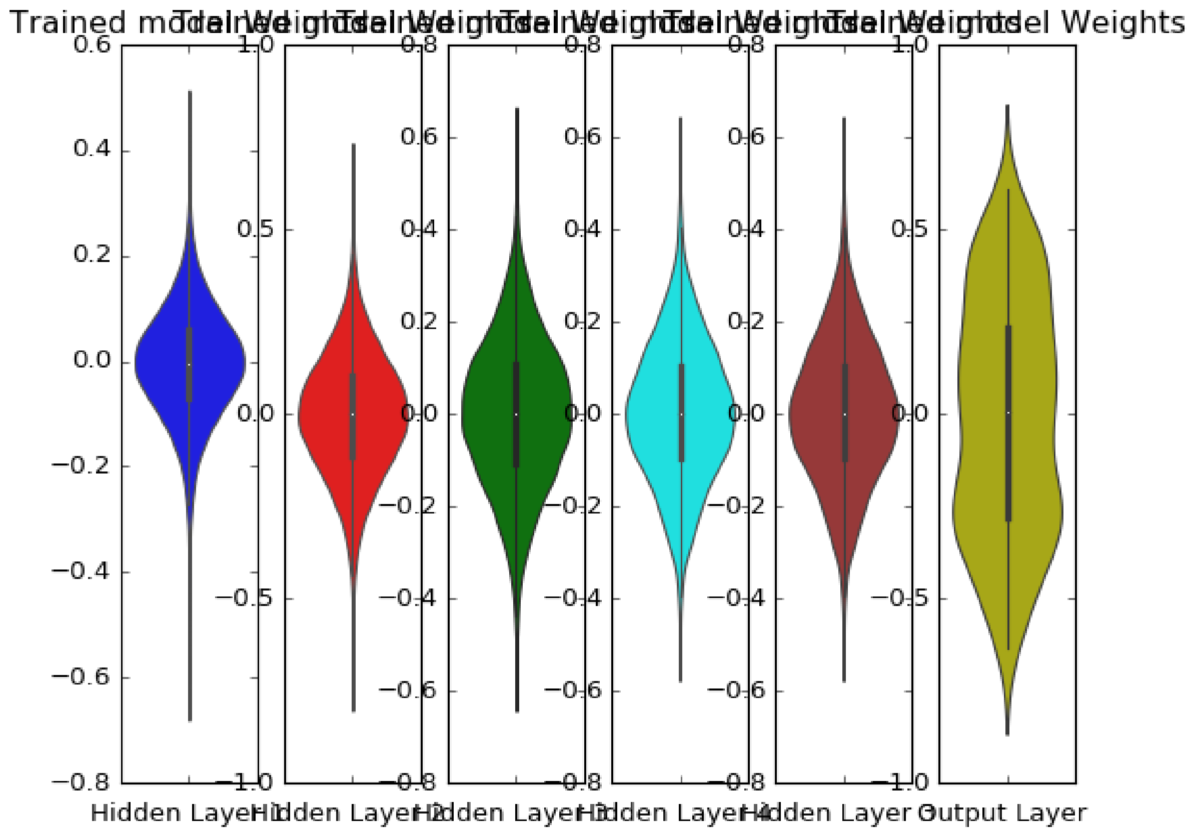
```
Test score: 0.08187933229937626
Test accuracy: 0.9814
```

In [85]:

```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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

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



In [86]:
```
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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + Dropout + AdamOptimizer with 5 hidden layers

In [87]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormaliza

from keras.layers import Dropout

model_drop = Sequential()

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

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

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

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

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

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


model_drop.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_63 (Dense)             (None, 256)               200960
_________________________________________________________________
dropout_16 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_64 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_17 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_65 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_18 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_66 (Dense)             (None, 32)                2080
_________________________________________________________________
dropout_19 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_67 (Dense)             (None, 16)                528
_________________________________________________________________
dropout_20 (Dropout)         (None, 16)                0
_________________________________________________________________
dense_68 (Dense)             (None, 10)                170
=================================================================
Total params: 244,890
Trainable params: 244,890
Non-trainable params: 0
_________________________________________________________________
```

In [88]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 12s 200us/step - loss: 2.1425 -
acc: 0.2220 - val_loss: 1.5689 - val_acc: 0.5284
Epoch 2/20
60000/60000 [==============================] - 10s 160us/step - loss: 1.6576 -
acc: 0.4145 - val_loss: 1.3220 - val_acc: 0.5546
Epoch 3/20
60000/60000 [==============================] - 8s 138us/step - loss: 1.4285 - a
cc: 0.5034 - val_loss: 1.0335 - val_acc: 0.6939
Epoch 4/20
60000/60000 [==============================] - 8s 136us/step - loss: 1.2482 - a
cc: 0.5584 - val_loss: 0.8864 - val_acc: 0.7516
Epoch 5/20
60000/60000 [==============================] - 8s 137us/step - loss: 1.1219 - a
cc: 0.6141 - val_loss: 0.7616 - val_acc: 0.8076
Epoch 6/20
60000/60000 [==============================] - 8s 139us/step - loss: 1.0295 - a
cc: 0.6447 - val_loss: 0.6917 - val_acc: 0.8053
Epoch 7/20
60000/60000 [==============================] - 8s 135us/step - loss: 0.9681 - a
cc: 0.6627 - val_loss: 0.6602 - val_acc: 0.8036
Epoch 8/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.9128 - a
cc: 0.6807 - val_loss: 0.6569 - val_acc: 0.7593
Epoch 9/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.8891 - a
cc: 0.6880 - val_loss: 0.6218 - val_acc: 0.7908
Epoch 10/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.8644 - a
cc: 0.6985 - val_loss: 0.5864 - val_acc: 0.8127
Epoch 11/20
60000/60000 [==============================] - 8s 134us/step - loss: 0.8475 - a
cc: 0.7026 - val_loss: 0.5951 - val_acc: 0.7848
Epoch 12/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.8229 - a
cc: 0.7129 - val_loss: 0.5810 - val_acc: 0.8239
Epoch 13/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.7966 - a
cc: 0.7257 - val_loss: 0.5449 - val_acc: 0.8237
Epoch 14/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.7932 - a
cc: 0.7276 - val_loss: 0.5404 - val_acc: 0.8323
Epoch 15/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.7958 - a
cc: 0.7321 - val_loss: 0.5297 - val_acc: 0.8271
Epoch 16/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.7734 - a
cc: 0.7401 - val_loss: 0.5279 - val_acc: 0.8400
Epoch 17/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.7516 - a
cc: 0.7449 - val_loss: 0.5365 - val_acc: 0.8304
Epoch 18/20
```

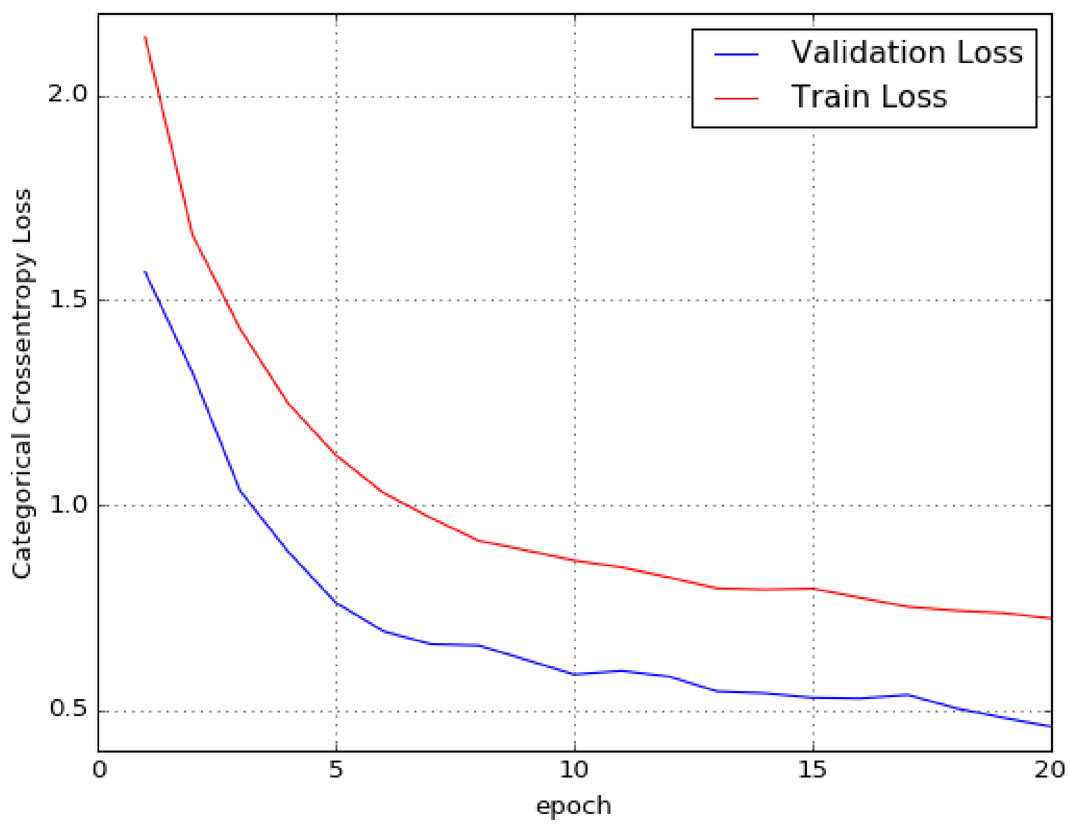
```
60000/60000 [==============================] - 9s 144us/step - loss: 0.7421 - a
cc: 0.7506 - val_loss: 0.5041 - val_acc: 0.8411
Epoch 19/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.7363 - a
cc: 0.7537 - val_loss: 0.4806 - val_acc: 0.8496
Epoch 20/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.7230 - a
cc: 0.7584 - val_loss: 0.4595 - val_acc: 0.8580
```

In [89]:

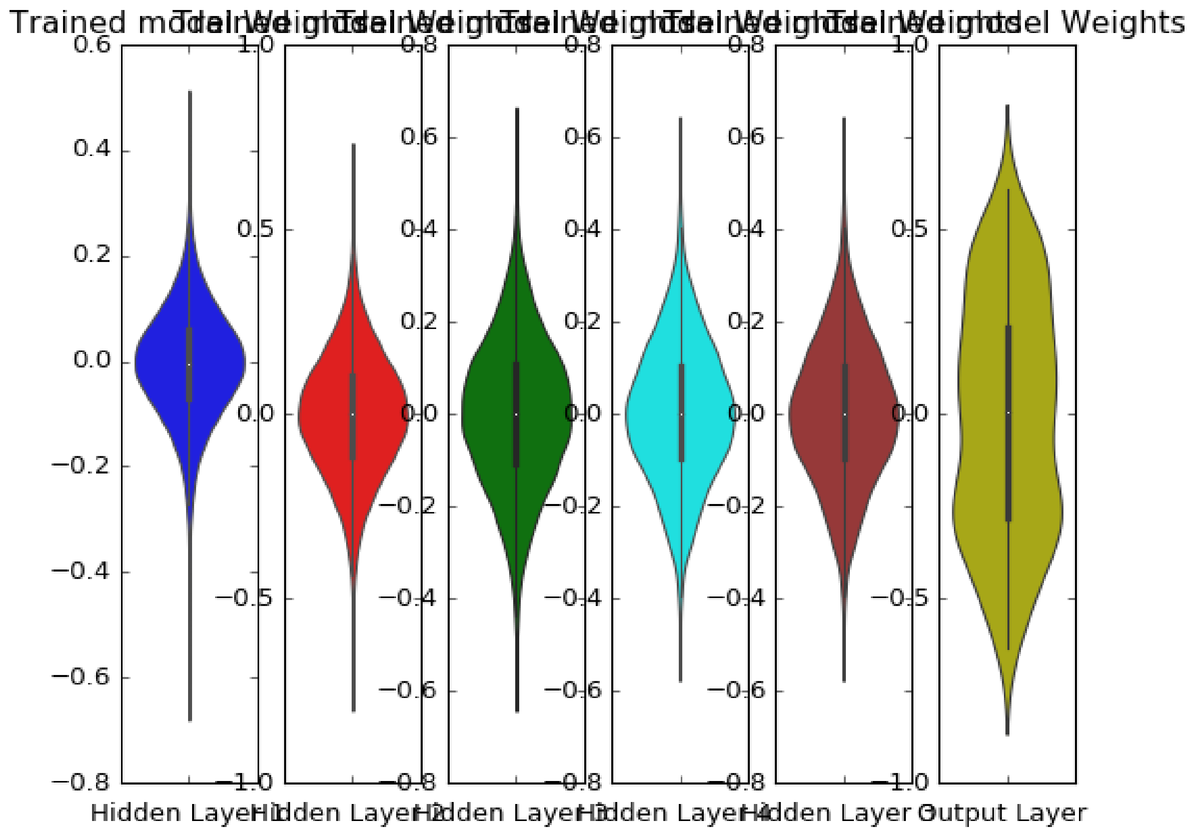
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
Test score: 0.4594558026790619
Test accuracy: 0.858
```

In [90]:
```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat
# 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

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



In [91]:
```
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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

MLP + BatchNormalization + Dropout + AdamOptimizer with 3 hidden layers

In [92]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormaliza

from keras.layers import Dropout

model_drop = Sequential()

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

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

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

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

model_drop.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=
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_69 (Dense)             (None, 256)               200960    
_________________________________________________________________
batch_normalization_21 (Batc (None, 256)               1024      
_________________________________________________________________
dropout_21 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_70 (Dense)             (None, 128)               32896     
_________________________________________________________________
batch_normalization_22 (Batc (None, 128)               512       
_________________________________________________________________
dropout_22 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_71 (Dense)             (None, 64)                8256      
_________________________________________________________________
batch_normalization_23 (Batc (None, 64)                256       
_________________________________________________________________
dropout_23 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_72 (Dense)             (None, 32)                2080      
_________________________________________________________________
batch_normalization_24 (Batc (None, 32)                128       
_________________________________________________________________
```

```
dropout_24 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_73 (Dense)             (None, 16)                528
_________________________________________________________________
batch_normalization_25 (Batc (None, 16)                64
_________________________________________________________________
dropout_25 (Dropout)         (None, 16)                0
_________________________________________________________________
dense_74 (Dense)             (None, 10)                170
=================================================================
Total params: 246,874
Trainable params: 245,882
Non-trainable params: 992
_________________________________________________________________
```

In [93]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 16s 271us/step - loss: 2.1278 -
acc: 0.2514 - val_loss: 1.1959 - val_acc: 0.7221
Epoch 2/20
60000/60000 [==============================] - 11s 176us/step - loss: 1.4928 -
acc: 0.4529 - val_loss: 0.7785 - val_acc: 0.7865
Epoch 3/20
60000/60000 [==============================] - 11s 184us/step - loss: 1.1906 -
acc: 0.5626 - val_loss: 0.5776 - val_acc: 0.8514
Epoch 4/20
60000/60000 [==============================] - 11s 185us/step - loss: 1.0123 -
acc: 0.6299 - val_loss: 0.4885 - val_acc: 0.8413
Epoch 5/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.9047 -
acc: 0.6725 - val_loss: 0.4304 - val_acc: 0.8450
Epoch 6/20
60000/60000 [==============================] - 11s 181us/step - loss: 0.8163 -
acc: 0.7062 - val_loss: 0.3703 - val_acc: 0.8852
Epoch 7/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.7419 - a
cc: 0.7370 - val_loss: 0.3397 - val_acc: 0.8845
Epoch 8/20
60000/60000 [==============================] - 10s 171us/step - loss: 0.6941 -
acc: 0.7580 - val_loss: 0.3168 - val_acc: 0.9081
Epoch 9/20
60000/60000 [==============================] - 11s 191us/step - loss: 0.6566 -
acc: 0.7774 - val_loss: 0.2958 - val_acc: 0.9234
Epoch 10/20
60000/60000 [==============================] - 10s 168us/step - loss: 0.6195 -
acc: 0.7901 - val_loss: 0.2795 - val_acc: 0.9047
Epoch 11/20
60000/60000 [==============================] - 10s 174us/step - loss: 0.5903 -
acc: 0.8044 - val_loss: 0.2705 - val_acc: 0.9022
Epoch 12/20
60000/60000 [==============================] - 10s 167us/step - loss: 0.5624 -
acc: 0.8158 - val_loss: 0.2521 - val_acc: 0.9353
Epoch 13/20
60000/60000 [==============================] - 12s 192us/step - loss: 0.5452 -
acc: 0.8250 - val_loss: 0.2466 - val_acc: 0.9243
Epoch 14/20
60000/60000 [==============================] - 12s 200us/step - loss: 0.5280 -
acc: 0.8308 - val_loss: 0.2431 - val_acc: 0.9507
Epoch 15/20
60000/60000 [==============================] - 12s 200us/step - loss: 0.5180 -
acc: 0.8363 - val_loss: 0.2258 - val_acc: 0.9539
Epoch 16/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.5051 -
acc: 0.8412 - val_loss: 0.2227 - val_acc: 0.9540
Epoch 17/20
60000/60000 [==============================] - 12s 197us/step - loss: 0.4821 -
acc: 0.8486 - val_loss: 0.2090 - val_acc: 0.9575
Epoch 18/20
```

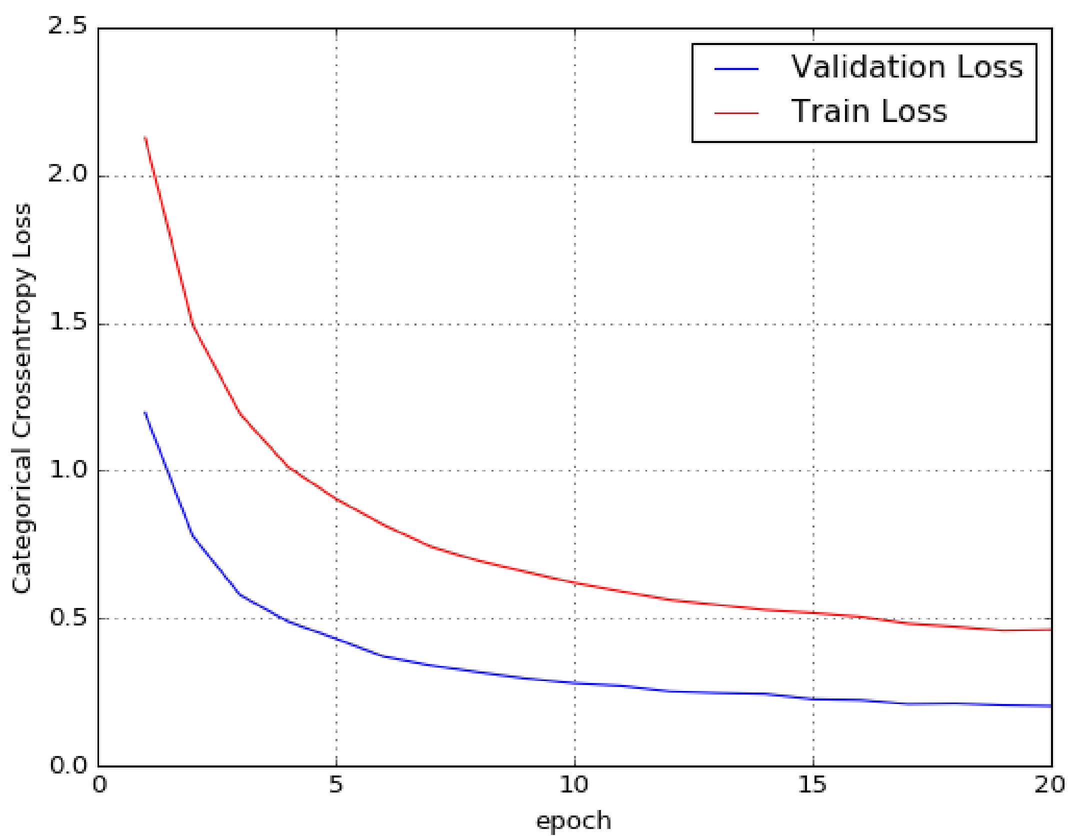
```
60000/60000 [==============================] - 12s 202us/step - loss: 0.4715 -
acc: 0.8532 - val_loss: 0.2110 - val_acc: 0.9592
Epoch 19/20
60000/60000 [==============================] - 12s 197us/step - loss: 0.4581 -
acc: 0.8579 - val_loss: 0.2048 - val_acc: 0.9599
Epoch 20/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.4610 -
acc: 0.8600 - val_loss: 0.2019 - val_acc: 0.9590
```

In [95]:

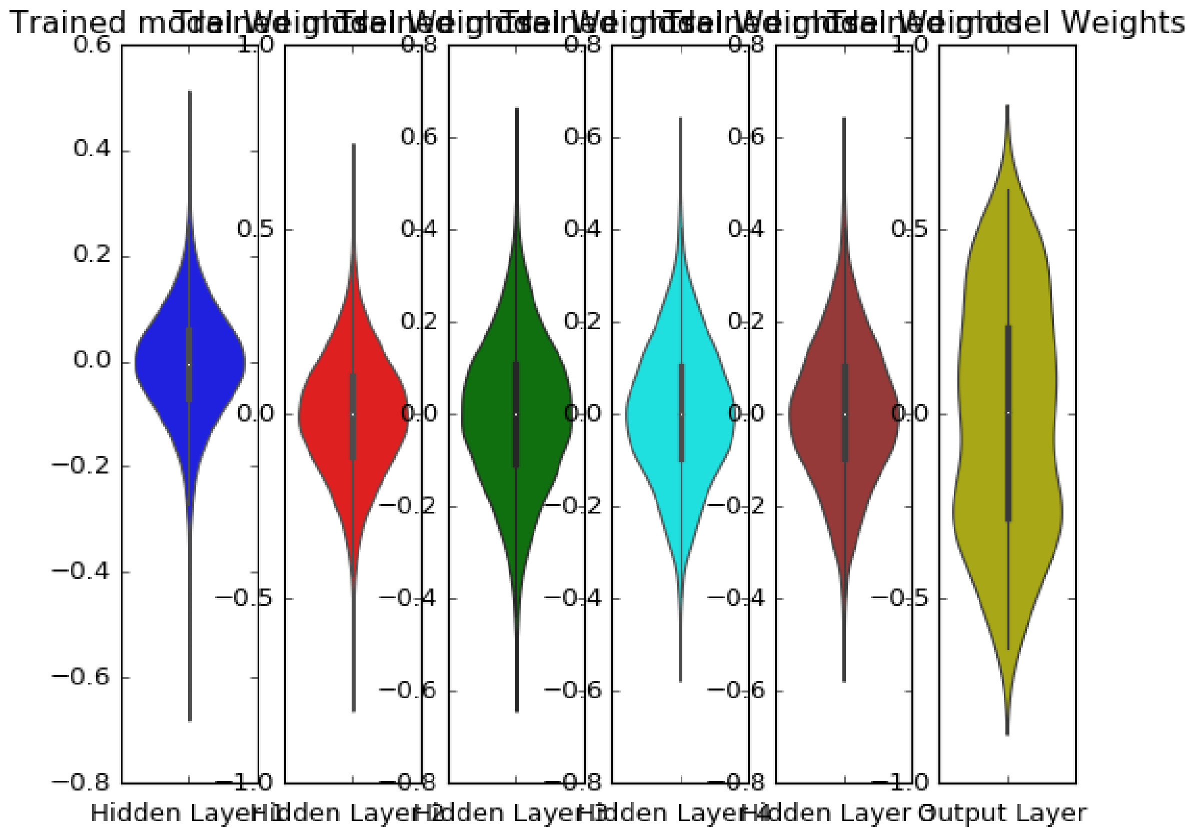
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
Test score: 0.2019003223091364
Test accuracy: 0.959
```

In [96]:
```
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_epo

# we will get val_loss and val_acc only when you pass the paramter validation_dat

# 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

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



In [97]:
```
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()
```




```
C:\Program Files\Anaconda3\lib\site-packages\scipy\stats\stats.py:1626: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is deprecated;
use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpr
eted as an array index, `arr[np.array(seq)]`, which will result either in an er
ror or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

In [98]:

```
#Comparisons
```

In [99]:
```
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", 97.44])
x.add_row(["2 layer", "with Batch Normalization", 97.98])
x.add_row(["2 layer", "with Dropuot and Batch Normalization ", 98.08])

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 Batch Normalization", 97.59])
x.add_row(["3 layer", "with Dropuot and Batch Normalization ", 97.84])

x.add_row(["5 layer", "without Dropout and Batch Normalization", 98.09])
x.add_row(["5 layer", "with Dropuot", 85.8])
x.add_row(["5 layer", "with Batch Normalization", 98.14])
x.add_row(["5 layer", "with Dropuot and Batch Normalization ", 95.9])

print(x)
```

```
+--------------+-----------------------------------------+----------+
| Architecture |                parameters               | Accuracy |
+--------------+-----------------------------------------+----------+
|   2 layer    | without Dropout and Batch Normalization |   97.9   |
|   2 layer    |               with Dropuot              |  97.44   |
|   2 layer    |         with Batch Normalization        |  97.98   |
|   2 layer    |  with Dropuot and Batch Normalization   |  98.08   |
|   3 layer    | without Dropout and Batch Normalization |  97.87   |
|   3 layer    |               with Dropuot              |  97.38   |
|   3 layer    |         with Batch Normalization        |  97.59   |
|   3 layer    |  with Dropuot and Batch Normalization   |  97.84   |
|   5 layer    | without Dropout and Batch Normalization |  98.09   |
|   5 layer    |               with Dropuot              |   85.8   |
|   5 layer    |         with Batch Normalization        |  98.14   |
|   5 layer    |  with Dropuot and Batch Normalization   |   95.9   |
+--------------+-----------------------------------------+----------+
```

In [ ]: