In [2]:

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

```
Using TensorFlow backend.
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorflow/python/f
ramework/dtypes.py:516: FutureWarning: Passing (type, 1) or '1type' as a syn
onym of type is deprecated; in a future version of numpy, it will be underst
ood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorflow/python/f
ramework/dtypes.py:517: FutureWarning: Passing (type, 1) or '1type' as a syn
onym of type is deprecated; in a future version of numpy, it will be underst
ood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorflow/python/f
ramework/dtypes.py:518: FutureWarning: Passing (type, 1) or '1type' as a syn
onym of type is deprecated; in a future version of numpy, it will be underst
ood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorflow/python/f
ramework/dtypes.py:519: FutureWarning: Passing (type, 1) or '1type' as a syn
onym of type is deprecated; in a future version of numpy, it will be underst
ood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorflow/python/f
ramework/dtypes.py:520: FutureWarning: Passing (type, 1) or '1type' as a syn
onym of type is deprecated; in a future version of numpy, it will be underst
ood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorflow/python/f
ramework/dtypes.py:525: FutureWarning: Passing (type, 1) or '1type' as a syn
onym of type is deprecated; in a future version of numpy, it will be underst
ood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorboard/compat/
tensorflow_stub/dtypes.py:541: FutureWarning: Passing (type, 1) or '1type' a
s a synonym of type is deprecated; in a future version of numpy, it will be
understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorboard/compat/
tensorflow_stub/dtypes.py:542: FutureWarning: Passing (type, 1) or '1type' a
s a synonym of type is deprecated; in a future version of numpy, it will be
understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorboard/compat/
tensorflow_stub/dtypes.py:543: FutureWarning: Passing (type, 1) or '1type' a
s a synonym of type is deprecated; in a future version of numpy, it will be
understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorboard/compat/
tensorflow_stub/dtypes.py:544: FutureWarning: Passing (type, 1) or '1type' a
s a synonym of type is deprecated; in a future version of numpy, it will be
understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorboard/compat/
tensorflow_stub/dtypes.py:545: FutureWarning: Passing (type, 1) or '1type' a
```

```
s a synonym of type is deprecated; in a future version of numpy, it will be
understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/tensorboard/compat/
tensorflow_stub/dtypes.py:550: FutureWarning: Passing (type, 1) or '1type' a
s a synonym of type is deprecated; in a future version of numpy, it will be
understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

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()
    # plt.show()
    # fig.canvas.draw()
    plt.show()
```

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 (http
s://s3.amazonaws.com/img-datasets/mnist.npz)
11493376/11490434 [==============================] - 0s 0us/step
```

In [5]:

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

```
Number of training examples : 60000 and each image is of shape (28, 28)
Number of testing examples : 10000 and each image is of shape (28, 28)
```

In [6]:

```
# if you observe the input shape its 2 dimensional vector
# for each image we have a (28*28) vector
# we will convert the (28*28) vector into single dimensional vector of 1 * 784

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

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

```
Number of training examples : 60000 and each image is of shape (784)
Number of testing 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 data
# 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 constructor

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

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

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

###

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

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

# Dense implements the operation: output = activation(dot(input, kernel) + bias) where
# activation is the element-wise activation function passed as the activation argument,
# kernel is a weights matrix created by the layer, and
# bias is a bias vector created by the layer (only applicable if use_bias is True).

# output = activation(dot(input, kernel) + bias)  => y = activation(WT. X + b)

####

# https://keras.io/activations/

# Activations can either be used through an Activation layer, or through the activation arg

# from keras.layers import Activation, Dense

# model.add(Dense(64))
# model.add(Activation('tanh'))

# This is equivalent to:
# model.add(Dense(64, activation='tanh'))

# there are many activation functions ar available ex: tanh, relu, softmax


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

In [13]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

MLP + ReLU + Adam: 2 Hidden Layers and without Dropout and Batch Normalization

In [14]:

```
model_relu = Sequential()
model_relu.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_relu.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

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

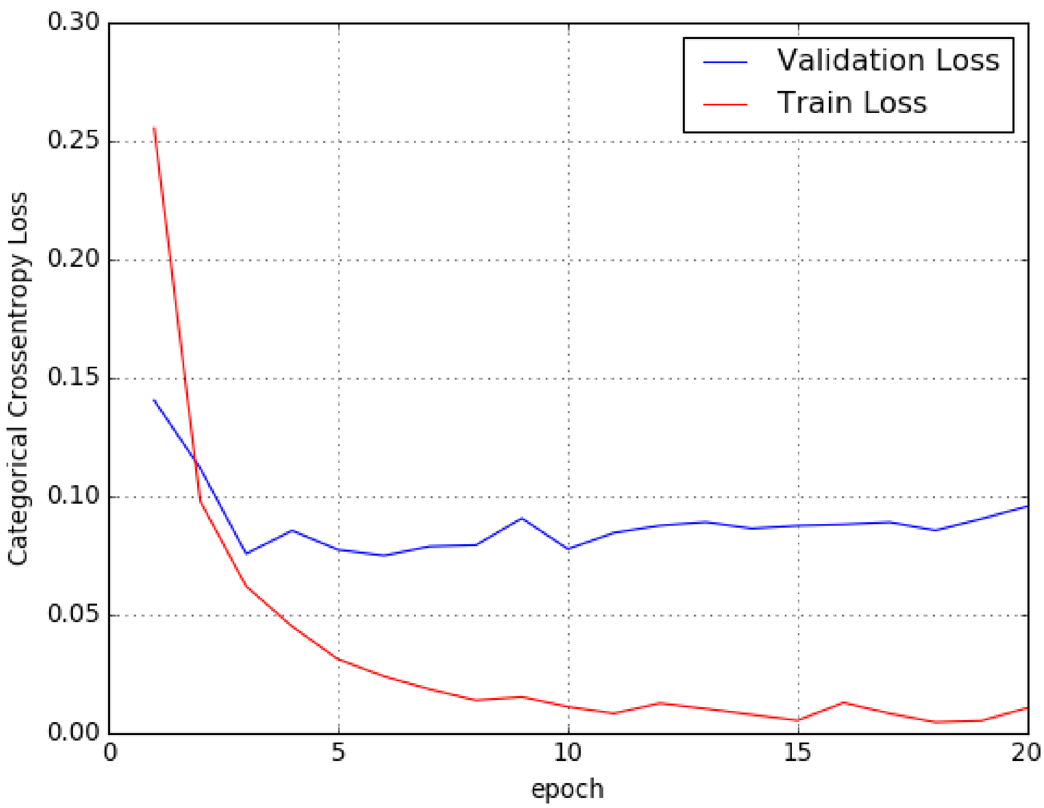
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 364)               285740
_________________________________________________________________
dense_2 (Dense)              (None, 52)                18980
_________________________________________________________________
dense_3 (Dense)              (None, 10)                530
=================================================================
Total params: 305,250
Trainable params: 305,250
Non-trainable params: 0
_________________________________________________________________
None
WARNING:tensorflow:From /home/komalumrethe/anaconda3/lib/python3.5/site-pa
ckages/keras/backend/tensorflow_backend.py:422: The name tf.global_variabl
es is deprecated. Please use tf.compat.v1.global_variables instead.

Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.2553
- accuracy: 0.9239 - val_loss: 0.1404 - val_accuracy: 0.9576
Epoch 2/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0980
- accuracy: 0.9707 - val_loss: 0.1119 - val_accuracy: 0.9661
Epoch 3/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0620
- accuracy: 0.9810 - val_loss: 0.0757 - val_accuracy: 0.9770
Epoch 4/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0450
- accuracy: 0.9859 - val_loss: 0.0855 - val_accuracy: 0.9742
Epoch 5/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0312
- accuracy: 0.9907 - val_loss: 0.0774 - val_accuracy: 0.9784
Epoch 6/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0240
- accuracy: 0.9926 - val_loss: 0.0749 - val_accuracy: 0.9773
Epoch 7/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0185
- accuracy: 0.9942 - val_loss: 0.0788 - val_accuracy: 0.9789
Epoch 8/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0139
- accuracy: 0.9961 - val_loss: 0.0794 - val_accuracy: 0.9783
Epoch 9/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0153
- accuracy: 0.9952 - val_loss: 0.0907 - val_accuracy: 0.9759
```

```
Epoch 10/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0112
- accuracy: 0.9968 - val_loss: 0.0777 - val_accuracy: 0.9795
Epoch 11/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.0083
- accuracy: 0.9976 - val_loss: 0.0845 - val_accuracy: 0.9799
Epoch 12/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0126
- accuracy: 0.9960 - val_loss: 0.0876 - val_accuracy: 0.9779
Epoch 13/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0103
- accuracy: 0.9966 - val_loss: 0.0890 - val_accuracy: 0.9786
Epoch 14/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.0078
- accuracy: 0.9976 - val_loss: 0.0864 - val_accuracy: 0.9797
Epoch 15/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0054
- accuracy: 0.9983 - val_loss: 0.0876 - val_accuracy: 0.9817
Epoch 16/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0129
- accuracy: 0.9957 - val_loss: 0.0881 - val_accuracy: 0.9813
Epoch 17/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0083
- accuracy: 0.9973 - val_loss: 0.0890 - val_accuracy: 0.9800
Epoch 18/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0046
- accuracy: 0.9985 - val_loss: 0.0856 - val_accuracy: 0.9815
Epoch 19/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0053
- accuracy: 0.9982 - val_loss: 0.0905 - val_accuracy: 0.9815
Epoch 20/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0106
- accuracy: 0.9968 - val_loss: 0.0957 - val_accuracy: 0.9792
```

In [15]:

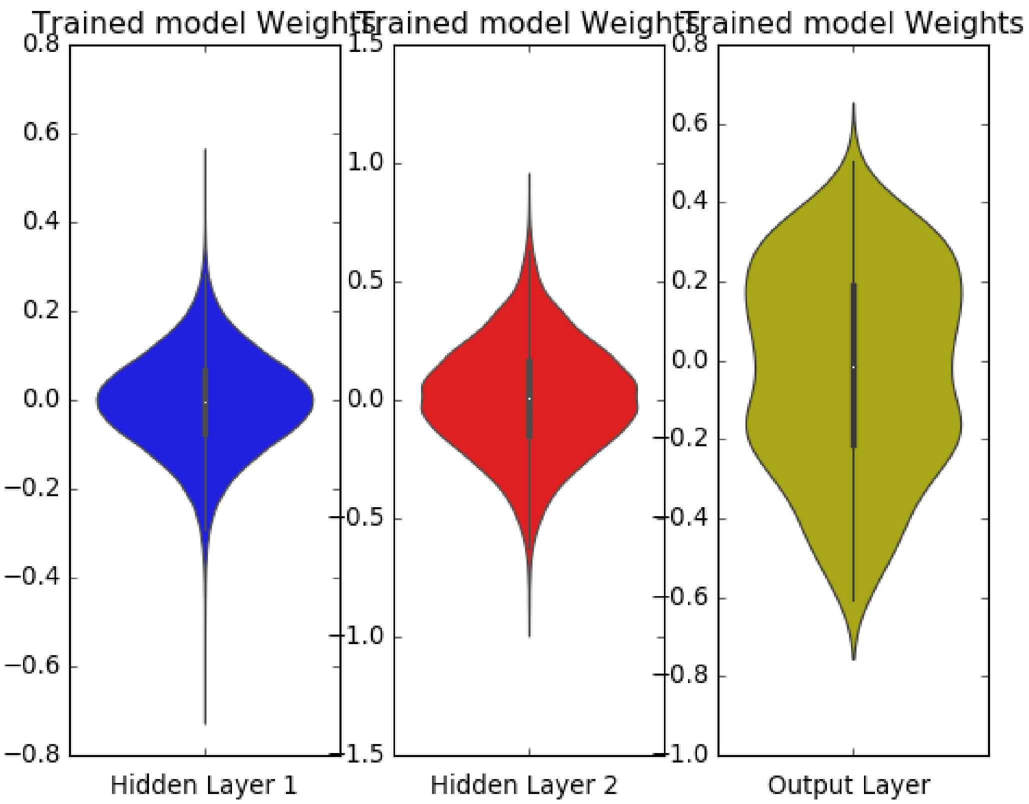
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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


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

```
Test score: 0.09574766628526103
Test accuracy: 0.979200005531311
```



In [16]:

```
w_after = model_relu.get_weights()

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


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

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

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



MLP + Batch-Norm on 2 hidden Layers + AdamOptimizer

In [17]:

```
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(364, activation='relu', input_shape=(input_dim,),
                      kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_batch.add(BatchNormalization())

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

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


model_batch.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_4 (Dense)              (None, 364)               285740    
_________________________________________________________________
batch_normalization_1 (Batch (None, 364)               1456      
_________________________________________________________________
dense_5 (Dense)              (None, 52)                18980     
_________________________________________________________________
batch_normalization_2 (Batch (None, 52)                208       
_________________________________________________________________
dense_6 (Dense)              (None, 10)                530       
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

In [18]:

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

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

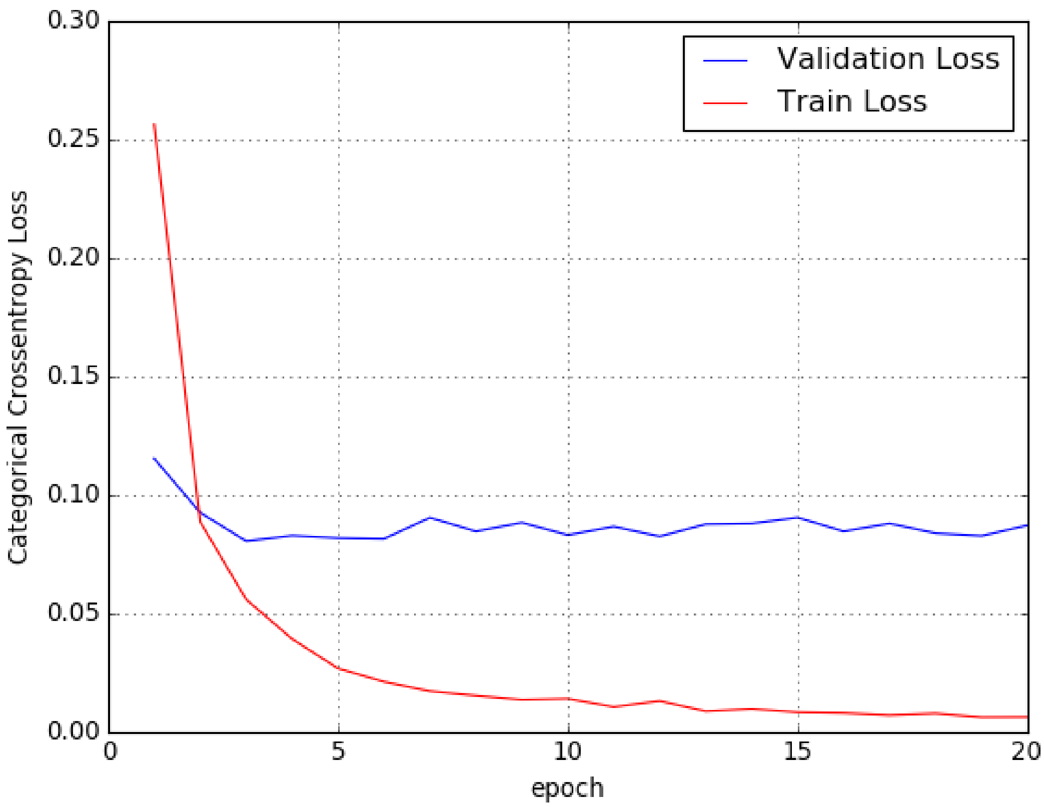
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.2564 -
accuracy: 0.9274 - val_loss: 0.1154 - val_accuracy: 0.9668
Epoch 2/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0888 -
accuracy: 0.9743 - val_loss: 0.0926 - val_accuracy: 0.9719
Epoch 3/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0560 -
accuracy: 0.9835 - val_loss: 0.0806 - val_accuracy: 0.9741
Epoch 4/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0393 -
accuracy: 0.9876 - val_loss: 0.0829 - val_accuracy: 0.9733
Epoch 5/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0269 -
accuracy: 0.9919 - val_loss: 0.0819 - val_accuracy: 0.9741
Epoch 6/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0212 -
accuracy: 0.9935 - val_loss: 0.0815 - val_accuracy: 0.9748
Epoch 7/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0173 -
accuracy: 0.9950 - val_loss: 0.0905 - val_accuracy: 0.9736
Epoch 8/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0154 -
accuracy: 0.9952 - val_loss: 0.0847 - val_accuracy: 0.9769
Epoch 9/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0137 -
accuracy: 0.9958 - val_loss: 0.0884 - val_accuracy: 0.9758
Epoch 10/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0141 -
accuracy: 0.9955 - val_loss: 0.0831 - val_accuracy: 0.9762
Epoch 11/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0106 -
accuracy: 0.9969 - val_loss: 0.0867 - val_accuracy: 0.9764
Epoch 12/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0131 -
accuracy: 0.9956 - val_loss: 0.0825 - val_accuracy: 0.9771
Epoch 13/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0088 -
accuracy: 0.9973 - val_loss: 0.0877 - val_accuracy: 0.9769
Epoch 14/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0097 -
accuracy: 0.9969 - val_loss: 0.0880 - val_accuracy: 0.9785
Epoch 15/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0084 -
accuracy: 0.9974 - val_loss: 0.0905 - val_accuracy: 0.9761
Epoch 16/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0080 -
accuracy: 0.9975 - val_loss: 0.0847 - val_accuracy: 0.9776
Epoch 17/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0071 -
accuracy: 0.9976 - val_loss: 0.0880 - val_accuracy: 0.9786
Epoch 18/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0079 -
```

```
accuracy: 0.9975 - val_loss: 0.0839 - val_accuracy: 0.9805
Epoch 19/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.0063 -
accuracy: 0.9980 - val_loss: 0.0828 - val_accuracy: 0.9792
Epoch 20/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0063 -
accuracy: 0.9980 - val_loss: 0.0872 - val_accuracy: 0.9796
```

In [19]:

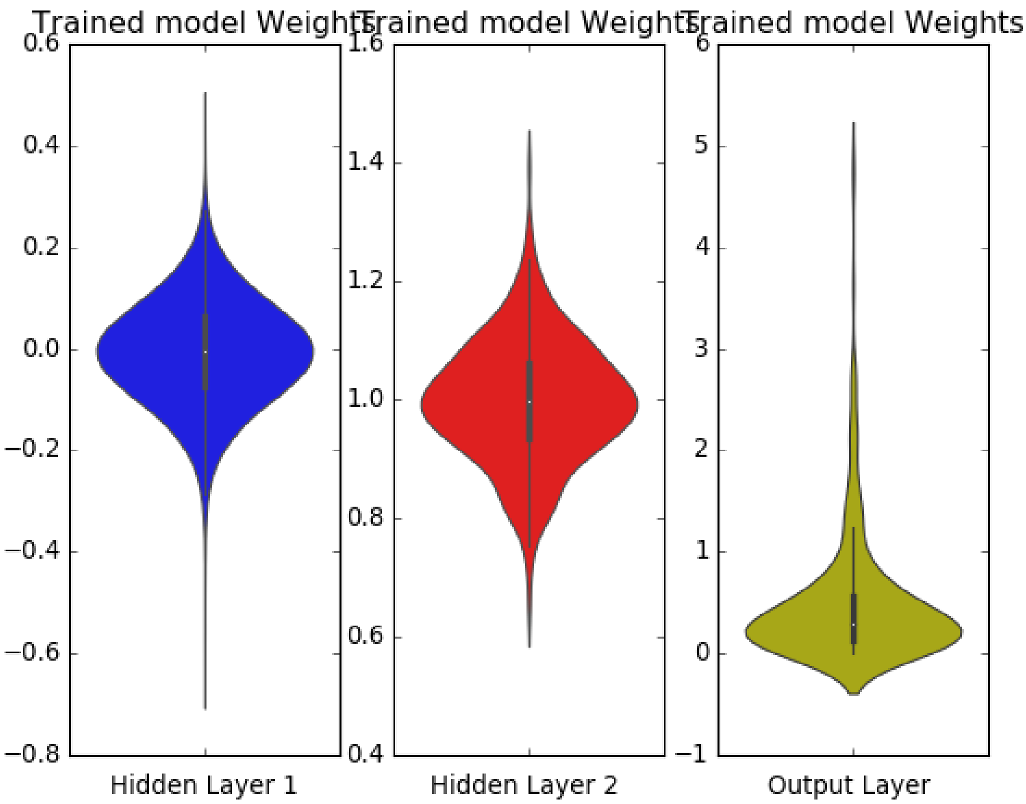
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.0872070386303676
Test accuracy: 0.9796000123023987
```



In [20]:

```
w_after = model_batch.get_weights()

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


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

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

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



MLP + Dropout (dropout rate = 0.5) on 2 hidden layers +

AdamOptimizer

In [21]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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

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


model_drop.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 364)               285740
_________________________________________________________________
batch_normalization_3 (Batch (None, 364)               1456
_________________________________________________________________
dropout_1 (Dropout)          (None, 364)               0
_________________________________________________________________
dense_8 (Dense)              (None, 52)                18980
_________________________________________________________________
batch_normalization_4 (Batch (None, 52)                208
_________________________________________________________________
dropout_2 (Dropout)          (None, 52)                0
_________________________________________________________________
dense_9 (Dense)              (None, 10)                530
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

In [22]:

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

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.6613 -
accuracy: 0.7983 - val_loss: 0.1981 - val_accuracy: 0.9418
Epoch 2/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.3129 -
accuracy: 0.9081 - val_loss: 0.1463 - val_accuracy: 0.9566
Epoch 3/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.2397 -
accuracy: 0.9315 - val_loss: 0.1147 - val_accuracy: 0.9666
Epoch 4/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.2024 -
accuracy: 0.9421 - val_loss: 0.1061 - val_accuracy: 0.9666
Epoch 5/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.1766 -
accuracy: 0.9488 - val_loss: 0.0959 - val_accuracy: 0.9709
Epoch 6/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1585 -
accuracy: 0.9534 - val_loss: 0.0912 - val_accuracy: 0.9731
Epoch 7/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1439 -
accuracy: 0.9577 - val_loss: 0.0841 - val_accuracy: 0.9746
Epoch 8/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.1366 -
accuracy: 0.9604 - val_loss: 0.0823 - val_accuracy: 0.9750
Epoch 9/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.1233 -
accuracy: 0.9642 - val_loss: 0.0749 - val_accuracy: 0.9764
Epoch 10/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.1192 -
accuracy: 0.9650 - val_loss: 0.0701 - val_accuracy: 0.9790
Epoch 11/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1116 -
accuracy: 0.9664 - val_loss: 0.0718 - val_accuracy: 0.9789
Epoch 12/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1080 -
accuracy: 0.9690 - val_loss: 0.0715 - val_accuracy: 0.9788
Epoch 13/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0971 -
accuracy: 0.9713 - val_loss: 0.0685 - val_accuracy: 0.9796
Epoch 14/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0969 -
accuracy: 0.9706 - val_loss: 0.0689 - val_accuracy: 0.9804
Epoch 15/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.0928 -
accuracy: 0.9726 - val_loss: 0.0695 - val_accuracy: 0.9801
Epoch 16/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0880 -
accuracy: 0.9736 - val_loss: 0.0690 - val_accuracy: 0.9799
Epoch 17/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0857 -
accuracy: 0.9744 - val_loss: 0.0652 - val_accuracy: 0.9806
Epoch 18/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0819 -
```

```
accuracy: 0.9756 - val_loss: 0.0671 - val_accuracy: 0.9818
Epoch 19/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0808 -
accuracy: 0.9759 - val_loss: 0.0657 - val_accuracy: 0.9803
Epoch 20/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0751 -
accuracy: 0.9775 - val_loss: 0.0625 - val_accuracy: 0.9828
```

In [ ]:

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

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

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

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

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

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

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

MLP + Dropout (dropout rate = 0.25) on 2 hidden layers + AdamOptimizer

In [24]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.25))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.25))

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


model_drop.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_13 (Dense)             (None, 364)               285740
_________________________________________________________________
batch_normalization_7 (Batch (None, 364)               1456
_________________________________________________________________
dropout_5 (Dropout)          (None, 364)               0
_________________________________________________________________
dense_14 (Dense)             (None, 52)                18980
_________________________________________________________________
batch_normalization_8 (Batch (None, 52)                208
_________________________________________________________________
dropout_6 (Dropout)          (None, 52)                0
_________________________________________________________________
dense_15 (Dense)             (None, 10)                530
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

In [25]:

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

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.3875 -
accuracy: 0.8858 - val_loss: 0.1491 - val_accuracy: 0.9571
Epoch 2/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1723 -
accuracy: 0.9498 - val_loss: 0.1036 - val_accuracy: 0.9681
Epoch 3/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.1295 -
accuracy: 0.9614 - val_loss: 0.0851 - val_accuracy: 0.9737
Epoch 4/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.1073 -
accuracy: 0.9673 - val_loss: 0.0894 - val_accuracy: 0.9728
Epoch 5/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0900 -
accuracy: 0.9717 - val_loss: 0.0730 - val_accuracy: 0.9784
Epoch 6/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0771 -
accuracy: 0.9762 - val_loss: 0.0706 - val_accuracy: 0.9771
Epoch 7/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0724 -
accuracy: 0.9771 - val_loss: 0.0718 - val_accuracy: 0.9779
Epoch 8/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0652 -
accuracy: 0.9793 - val_loss: 0.0682 - val_accuracy: 0.9783
Epoch 9/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.0565 -
accuracy: 0.9819 - val_loss: 0.0669 - val_accuracy: 0.9807
Epoch 10/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0546 -
accuracy: 0.9827 - val_loss: 0.0658 - val_accuracy: 0.9805
Epoch 11/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0483 -
accuracy: 0.9842 - val_loss: 0.0643 - val_accuracy: 0.9810
Epoch 12/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0461 -
accuracy: 0.9854 - val_loss: 0.0673 - val_accuracy: 0.9815
Epoch 13/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0418 -
accuracy: 0.9862 - val_loss: 0.0693 - val_accuracy: 0.9805
Epoch 14/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0403 -
accuracy: 0.9868 - val_loss: 0.0652 - val_accuracy: 0.9809
Epoch 15/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0389 -
accuracy: 0.9872 - val_loss: 0.0608 - val_accuracy: 0.9817
Epoch 16/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0373 -
accuracy: 0.9876 - val_loss: 0.0585 - val_accuracy: 0.9836
Epoch 17/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0368 -
accuracy: 0.9876 - val_loss: 0.0629 - val_accuracy: 0.9832
Epoch 18/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0325 -
```

```
accuracy: 0.9892 - val_loss: 0.0611 - val_accuracy: 0.9824
Epoch 19/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0294 -
accuracy: 0.9902 - val_loss: 0.0637 - val_accuracy: 0.9826
Epoch 20/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0292 -
accuracy: 0.9907 - val_loss: 0.0636 - val_accuracy: 0.9829
```

MLP + Dropout (dropout rate = 0.75) on 2 hidden layers + AdamOptimizer

In [27]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.25))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.25))

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


model_drop.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_16 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_9 (Batch (None, 364)               1456      
_________________________________________________________________
dropout_7 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_17 (Dense)             (None, 52)                18980     
_________________________________________________________________
batch_normalization_10 (Batc (None, 52)                208       
_________________________________________________________________
dropout_8 (Dropout)          (None, 52)                0         
_________________________________________________________________
dense_18 (Dense)             (None, 10)                530       
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

In [28]:

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

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

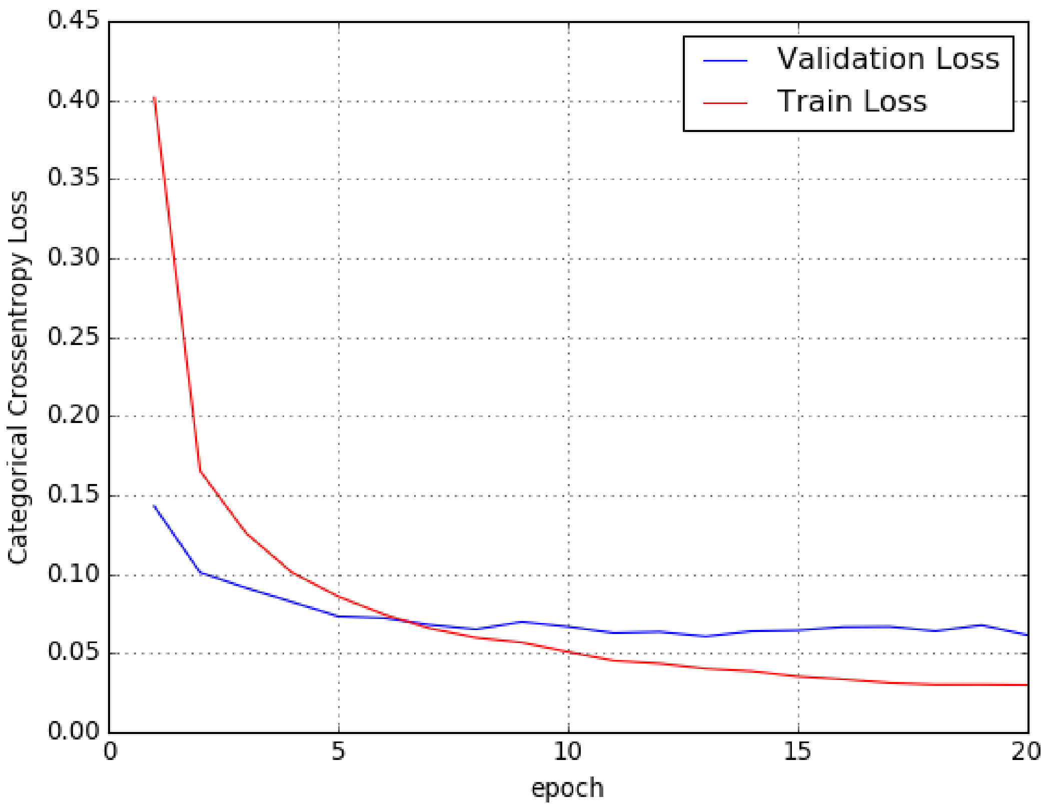
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.4016 -
accuracy: 0.8825 - val_loss: 0.1430 - val_accuracy: 0.9583
Epoch 2/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1653 -
accuracy: 0.9513 - val_loss: 0.1010 - val_accuracy: 0.9689
Epoch 3/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1259 -
accuracy: 0.9630 - val_loss: 0.0912 - val_accuracy: 0.9723
Epoch 4/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1010 -
accuracy: 0.9685 - val_loss: 0.0823 - val_accuracy: 0.9741
Epoch 5/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0858 -
accuracy: 0.9734 - val_loss: 0.0733 - val_accuracy: 0.9782
Epoch 6/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0745 -
accuracy: 0.9769 - val_loss: 0.0722 - val_accuracy: 0.9794
Epoch 7/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0656 -
accuracy: 0.9792 - val_loss: 0.0679 - val_accuracy: 0.9793
Epoch 8/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0598 -
accuracy: 0.9807 - val_loss: 0.0650 - val_accuracy: 0.9813
Epoch 9/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0567 -
accuracy: 0.9817 - val_loss: 0.0698 - val_accuracy: 0.9795
Epoch 10/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0508 -
accuracy: 0.9839 - val_loss: 0.0667 - val_accuracy: 0.9813
Epoch 11/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0452 -
accuracy: 0.9857 - val_loss: 0.0627 - val_accuracy: 0.9822
Epoch 12/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0434 -
accuracy: 0.9857 - val_loss: 0.0634 - val_accuracy: 0.9821
Epoch 13/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0402 -
accuracy: 0.9870 - val_loss: 0.0605 - val_accuracy: 0.9815
Epoch 14/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0385 -
accuracy: 0.9869 - val_loss: 0.0638 - val_accuracy: 0.9822
Epoch 15/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0352 -
accuracy: 0.9883 - val_loss: 0.0644 - val_accuracy: 0.9825
Epoch 16/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0334 -
accuracy: 0.9892 - val_loss: 0.0665 - val_accuracy: 0.9812
Epoch 17/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0311 -
accuracy: 0.9899 - val_loss: 0.0667 - val_accuracy: 0.9829
Epoch 18/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0301 -
```

```
accuracy: 0.9901 - val_loss: 0.0639 - val_accuracy: 0.9829
Epoch 19/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0301 -
accuracy: 0.9899 - val_loss: 0.0676 - val_accuracy: 0.9821
Epoch 20/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0299 -
accuracy: 0.9898 - val_loss: 0.0614 - val_accuracy: 0.9835
```

In [29]:

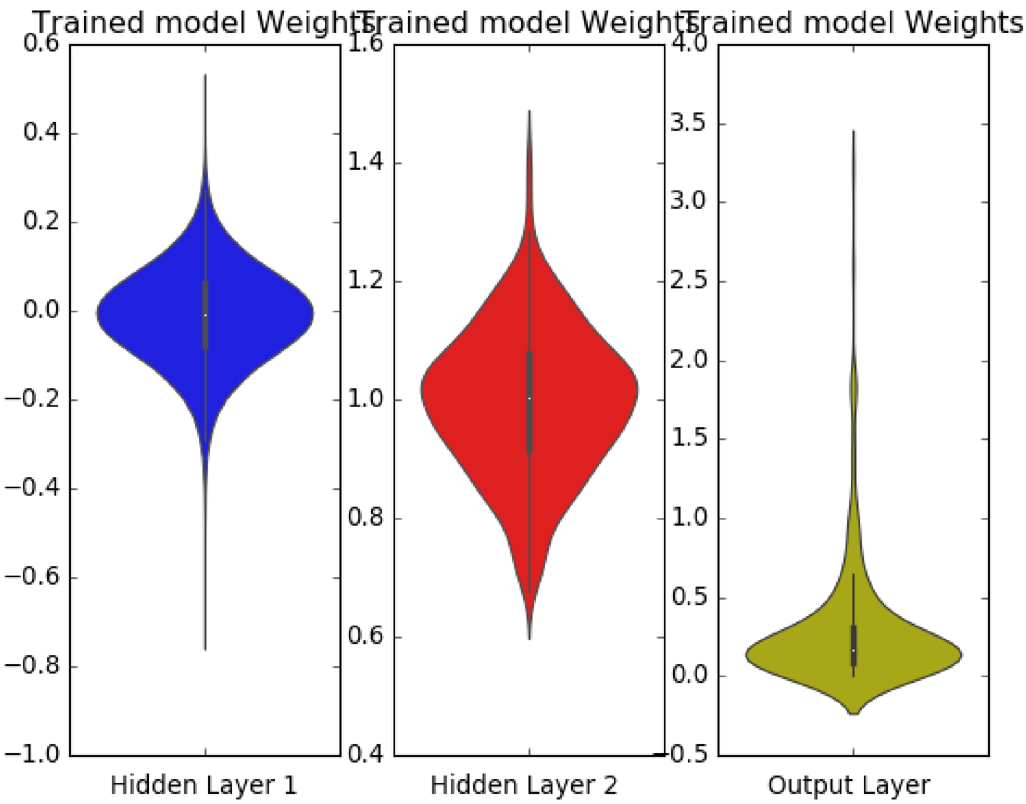
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.061442252652550815
Test accuracy: 0.9835000038146973
```



In [30]:

```
w_after = model_drop.get_weights()

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


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

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

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



MLP + ReLU + ADAM with 3 hidden layers without Dropout and

Batch Normalisation

In [31]:

```
model_relu = Sequential()
model_relu.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_relu.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

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

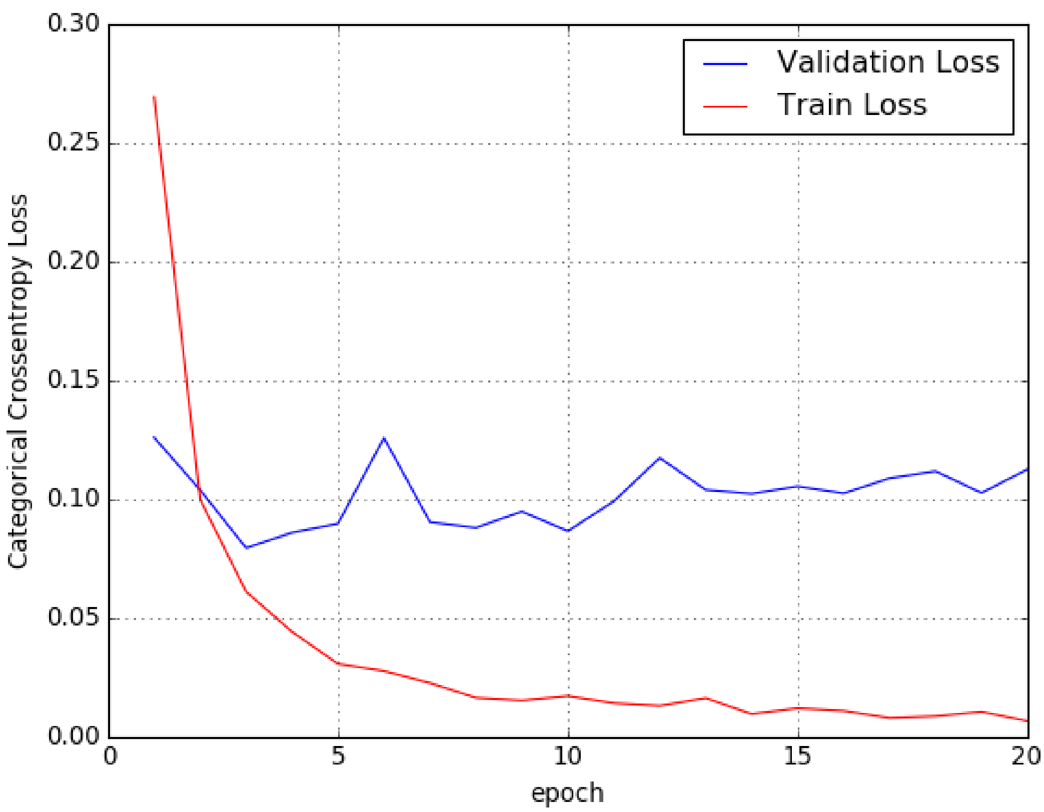
```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_19 (Dense)             (None, 364)               285740
_________________________________________________________________
dense_20 (Dense)             (None, 128)               46720
_________________________________________________________________
dense_21 (Dense)             (None, 52)                6708
_________________________________________________________________
dense_22 (Dense)             (None, 10)                530
=================================================================
Total params: 339,698
Trainable params: 339,698
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.2693 -
accuracy: 0.9198 - val_loss: 0.1260 - val_accuracy: 0.9616
Epoch 2/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0999 -
accuracy: 0.9691 - val_loss: 0.1039 - val_accuracy: 0.9661
Epoch 3/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0611 -
accuracy: 0.9812 - val_loss: 0.0795 - val_accuracy: 0.9751
Epoch 4/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0442 -
accuracy: 0.9858 - val_loss: 0.0860 - val_accuracy: 0.9731
Epoch 5/20
60000/60000 [==============================] - 3s 58us/step - loss: 0.0307 -
accuracy: 0.9900 - val_loss: 0.0898 - val_accuracy: 0.9751
Epoch 6/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0277 -
accuracy: 0.9909 - val_loss: 0.1259 - val_accuracy: 0.9657
Epoch 7/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0227 -
accuracy: 0.9923 - val_loss: 0.0904 - val_accuracy: 0.9763
Epoch 8/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0165 -
accuracy: 0.9947 - val_loss: 0.0880 - val_accuracy: 0.9757
Epoch 9/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0153 -
accuracy: 0.9951 - val_loss: 0.0949 - val_accuracy: 0.9758
```

```
Epoch 10/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0172 -
accuracy: 0.9946 - val_loss: 0.0866 - val_accuracy: 0.9773
Epoch 11/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0142 -
accuracy: 0.9952 - val_loss: 0.0991 - val_accuracy: 0.9766
Epoch 12/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0131 -
accuracy: 0.9956 - val_loss: 0.1174 - val_accuracy: 0.9758
Epoch 13/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0163 -
accuracy: 0.9949 - val_loss: 0.1040 - val_accuracy: 0.9783
Epoch 14/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0097 -
accuracy: 0.9965 - val_loss: 0.1023 - val_accuracy: 0.9775
Epoch 15/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0121 -
accuracy: 0.9962 - val_loss: 0.1054 - val_accuracy: 0.9767
Epoch 16/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.0110 -
accuracy: 0.9963 - val_loss: 0.1025 - val_accuracy: 0.9787
Epoch 17/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0080 -
accuracy: 0.9973 - val_loss: 0.1088 - val_accuracy: 0.9776
Epoch 18/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.0087 -
accuracy: 0.9973 - val_loss: 0.1118 - val_accuracy: 0.9769
Epoch 19/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.0106 -
accuracy: 0.9964 - val_loss: 0.1027 - val_accuracy: 0.9794
Epoch 20/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0067 -
accuracy: 0.9976 - val_loss: 0.1126 - val_accuracy: 0.9791
```

In [32]:

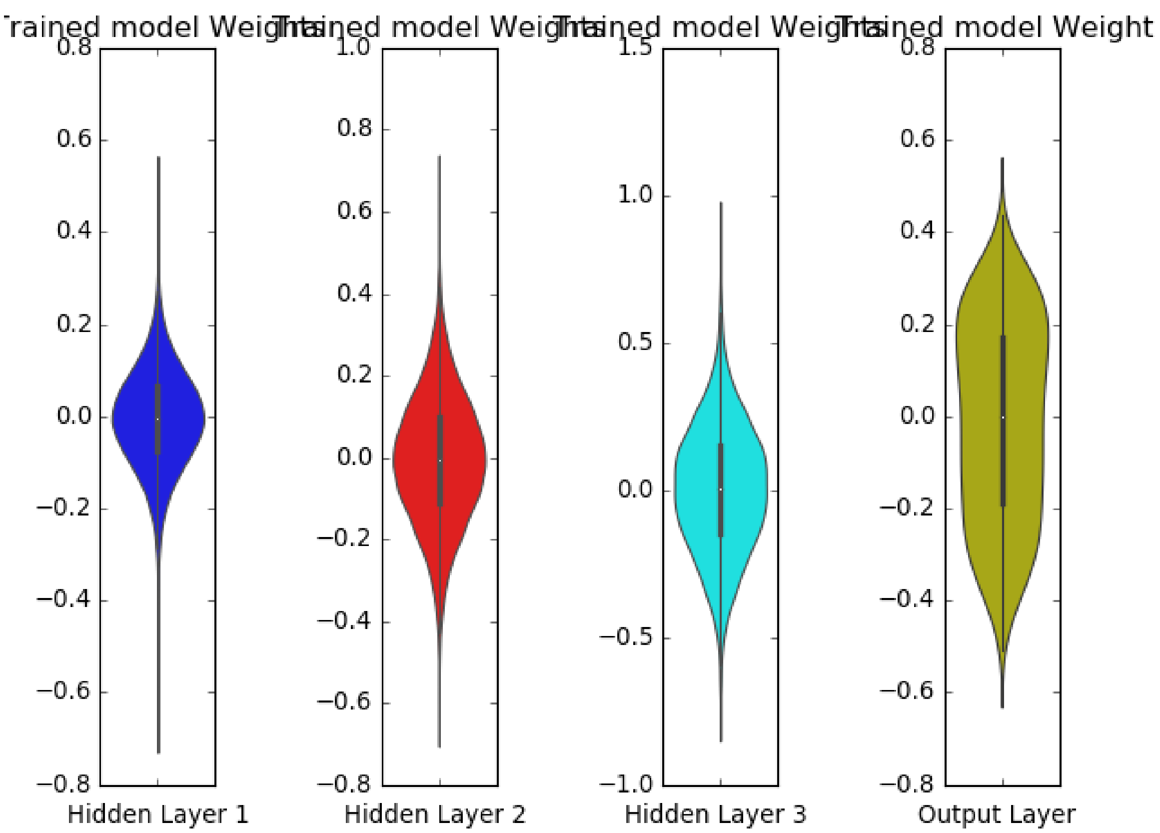
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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


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

```
Test score: 0.11258738235081656
Test accuracy: 0.9790999889373779
```



In [35]:

```
w_after = model_relu.get_weights()

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


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

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

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


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



MLP + ReLU + ADAM with 3 hidden layers with Batch Normalisation

In [36]:

```
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(364, activation='relu', input_shape=(input_dim,),
                      kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_batch.add(BatchNormalization())

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

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

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


model_batch.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 364)               285740
_________________________________________________________________
batch_normalization_11 (Batc (None, 364)               1456
_________________________________________________________________
dense_24 (Dense)             (None, 128)               46720
_________________________________________________________________
batch_normalization_12 (Batc (None, 128)               512
_________________________________________________________________
dense_25 (Dense)             (None, 52)                6708
_________________________________________________________________
batch_normalization_13 (Batc (None, 52)                208
_________________________________________________________________
dense_26 (Dense)             (None, 10)                530
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

In [37]:

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

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

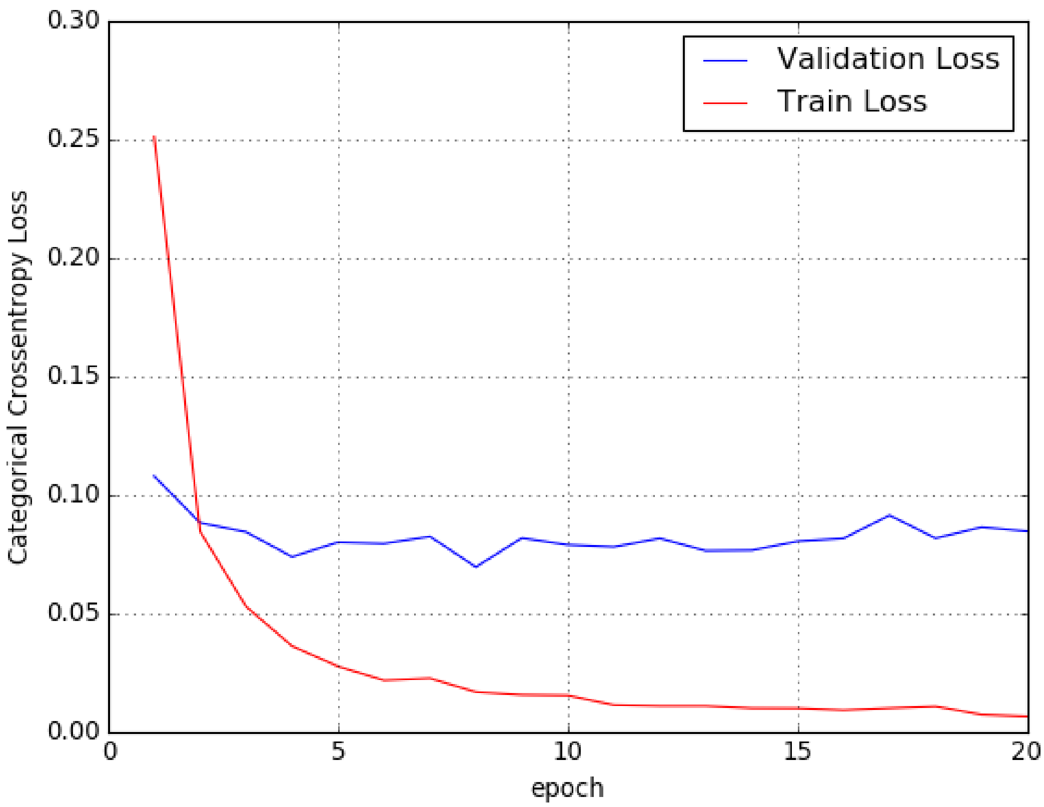
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.2511
- accuracy: 0.9278 - val_loss: 0.1079 - val_accuracy: 0.9677
Epoch 2/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0846
- accuracy: 0.9751 - val_loss: 0.0883 - val_accuracy: 0.9731
Epoch 3/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0529
- accuracy: 0.9836 - val_loss: 0.0844 - val_accuracy: 0.9727
Epoch 4/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0363
- accuracy: 0.9890 - val_loss: 0.0739 - val_accuracy: 0.9762
Epoch 5/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0277
- accuracy: 0.9913 - val_loss: 0.0801 - val_accuracy: 0.9765
Epoch 6/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0218
- accuracy: 0.9936 - val_loss: 0.0795 - val_accuracy: 0.9765
Epoch 7/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0227
- accuracy: 0.9924 - val_loss: 0.0825 - val_accuracy: 0.9775
Epoch 8/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0169
- accuracy: 0.9944 - val_loss: 0.0696 - val_accuracy: 0.9796
Epoch 9/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0157
- accuracy: 0.9949 - val_loss: 0.0819 - val_accuracy: 0.9760
Epoch 10/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0155
- accuracy: 0.9946 - val_loss: 0.0790 - val_accuracy: 0.9780
Epoch 11/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0114
- accuracy: 0.9962 - val_loss: 0.0781 - val_accuracy: 0.9788
Epoch 12/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0111
- accuracy: 0.9963 - val_loss: 0.0817 - val_accuracy: 0.9788
Epoch 13/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0110
- accuracy: 0.9964 - val_loss: 0.0765 - val_accuracy: 0.9801
Epoch 14/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0101
- accuracy: 0.9969 - val_loss: 0.0767 - val_accuracy: 0.9799
Epoch 15/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0100
- accuracy: 0.9967 - val_loss: 0.0805 - val_accuracy: 0.9788
Epoch 16/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0092
- accuracy: 0.9966 - val_loss: 0.0817 - val_accuracy: 0.9794
Epoch 17/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0101
- accuracy: 0.9967 - val_loss: 0.0915 - val_accuracy: 0.9770
Epoch 18/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.0108
```

```
- accuracy: 0.9962 - val_loss: 0.0818 - val_accuracy: 0.9786
Epoch 19/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0074
- accuracy: 0.9977 - val_loss: 0.0864 - val_accuracy: 0.9802
Epoch 20/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0066
- accuracy: 0.9979 - val_loss: 0.0847 - val_accuracy: 0.9790
```

In [38]:

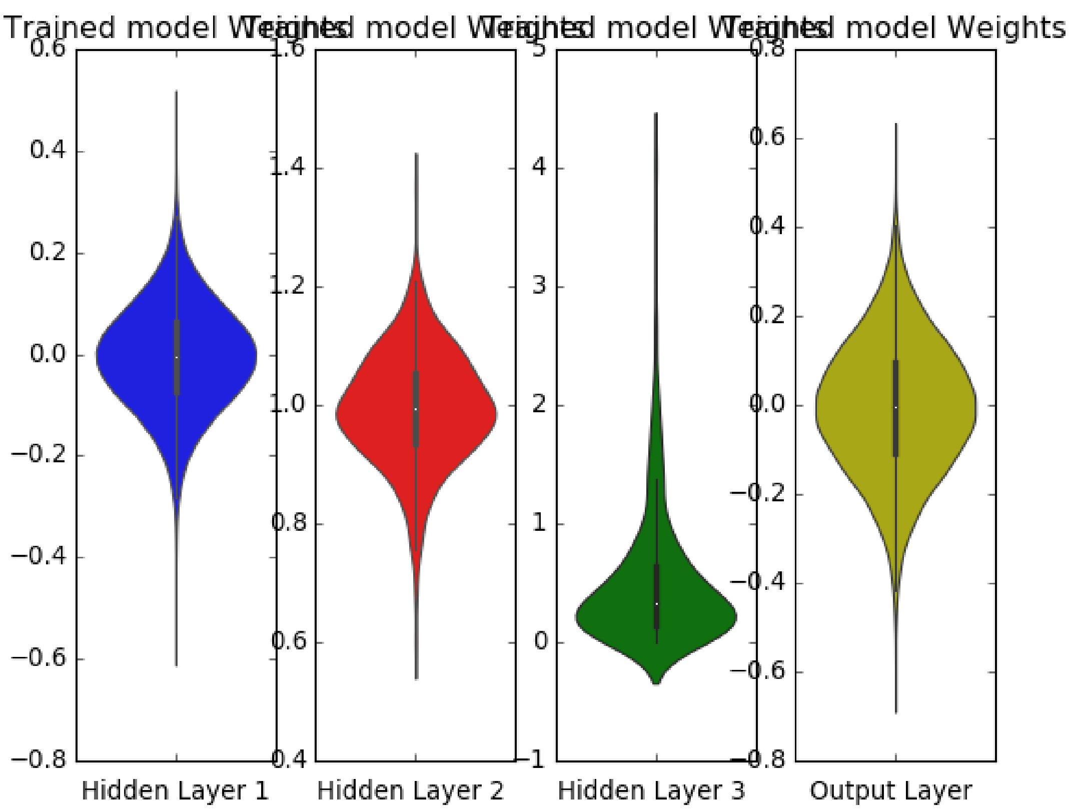
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.08473182359161073
Test accuracy: 0.9789999723434448
```



In [39]:

```
w_after = model_batch.get_weights()

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

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

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

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

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



MLP + ReLU + ADAM with 3 hidden layers with Dropout (dropout rate = 0.5)

In [40]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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

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


model_drop.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_27 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_14 (Batc (None, 364)               1456      
_________________________________________________________________
dropout_9 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_28 (Dense)             (None, 128)               46720     
_________________________________________________________________
batch_normalization_15 (Batc (None, 128)               512       
_________________________________________________________________
dropout_10 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_29 (Dense)             (None, 52)                6708      
_________________________________________________________________
batch_normalization_16 (Batc (None, 52)                208       
_________________________________________________________________
dropout_11 (Dropout)         (None, 52)                0         
_________________________________________________________________
dense_30 (Dense)             (None, 10)                530       
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

In [41]:

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

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

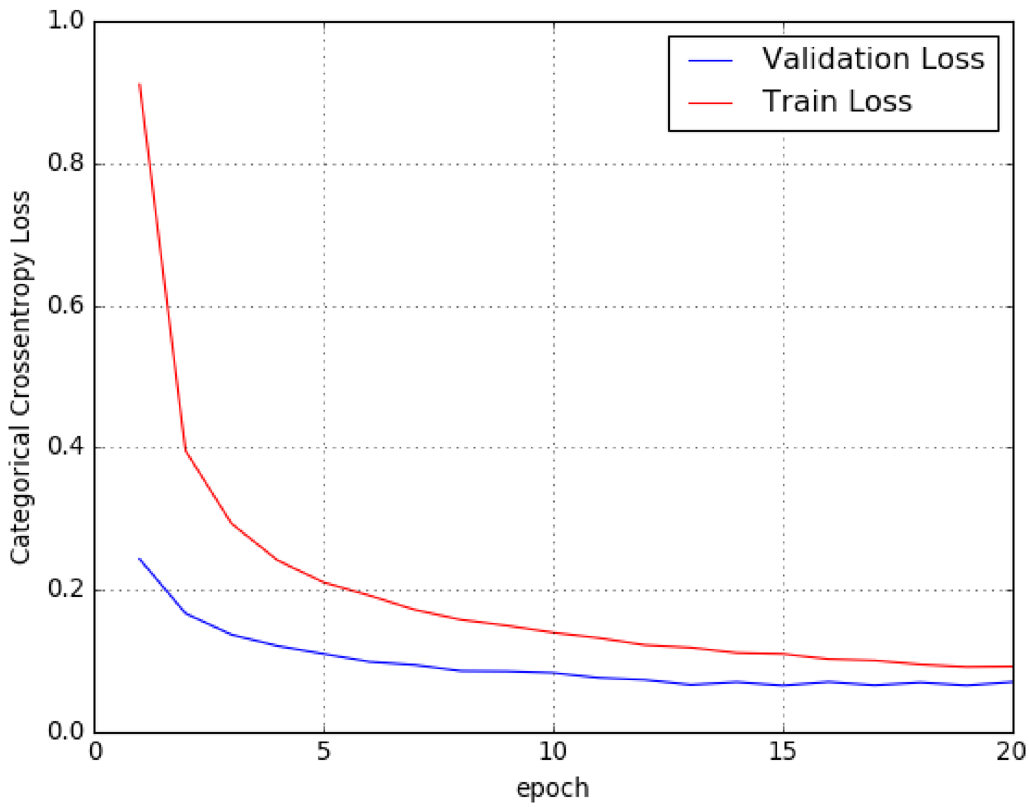
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.910
8 - accuracy: 0.7163 - val_loss: 0.2433 - val_accuracy: 0.9255
Epoch 2/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.3957
- accuracy: 0.8870 - val_loss: 0.1669 - val_accuracy: 0.9494
Epoch 3/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.2933
- accuracy: 0.9186 - val_loss: 0.1370 - val_accuracy: 0.9582
Epoch 4/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2417
- accuracy: 0.9342 - val_loss: 0.1212 - val_accuracy: 0.9634
Epoch 5/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.2106
- accuracy: 0.9423 - val_loss: 0.1101 - val_accuracy: 0.9682
Epoch 6/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1920
- accuracy: 0.9476 - val_loss: 0.0988 - val_accuracy: 0.9719
Epoch 7/20
60000/60000 [==============================] - 5s 92us/step - loss: 0.1718
- accuracy: 0.9532 - val_loss: 0.0944 - val_accuracy: 0.9735
Epoch 8/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1580
- accuracy: 0.9564 - val_loss: 0.0860 - val_accuracy: 0.9752
Epoch 9/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1497
- accuracy: 0.9588 - val_loss: 0.0855 - val_accuracy: 0.9754
Epoch 10/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1399
- accuracy: 0.9610 - val_loss: 0.0832 - val_accuracy: 0.9767
Epoch 11/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1323
- accuracy: 0.9631 - val_loss: 0.0762 - val_accuracy: 0.9782
Epoch 12/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1224
- accuracy: 0.9671 - val_loss: 0.0734 - val_accuracy: 0.9795
Epoch 13/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1188
- accuracy: 0.9679 - val_loss: 0.0666 - val_accuracy: 0.9810
Epoch 14/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1115
- accuracy: 0.9688 - val_loss: 0.0703 - val_accuracy: 0.9797
Epoch 15/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1100
- accuracy: 0.9693 - val_loss: 0.0656 - val_accuracy: 0.9804
Epoch 16/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1026
- accuracy: 0.9718 - val_loss: 0.0705 - val_accuracy: 0.9807
Epoch 17/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.1006
- accuracy: 0.9713 - val_loss: 0.0659 - val_accuracy: 0.9813
Epoch 18/20
```

```
60000/60000 [==============================] - 6s 93us/step - loss: 0.0951
- accuracy: 0.9729 - val_loss: 0.0698 - val_accuracy: 0.9803
Epoch 19/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0917
- accuracy: 0.9747 - val_loss: 0.0657 - val_accuracy: 0.9808
Epoch 20/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0924
- accuracy: 0.9749 - val_loss: 0.0704 - val_accuracy: 0.9825
```

In [42]:

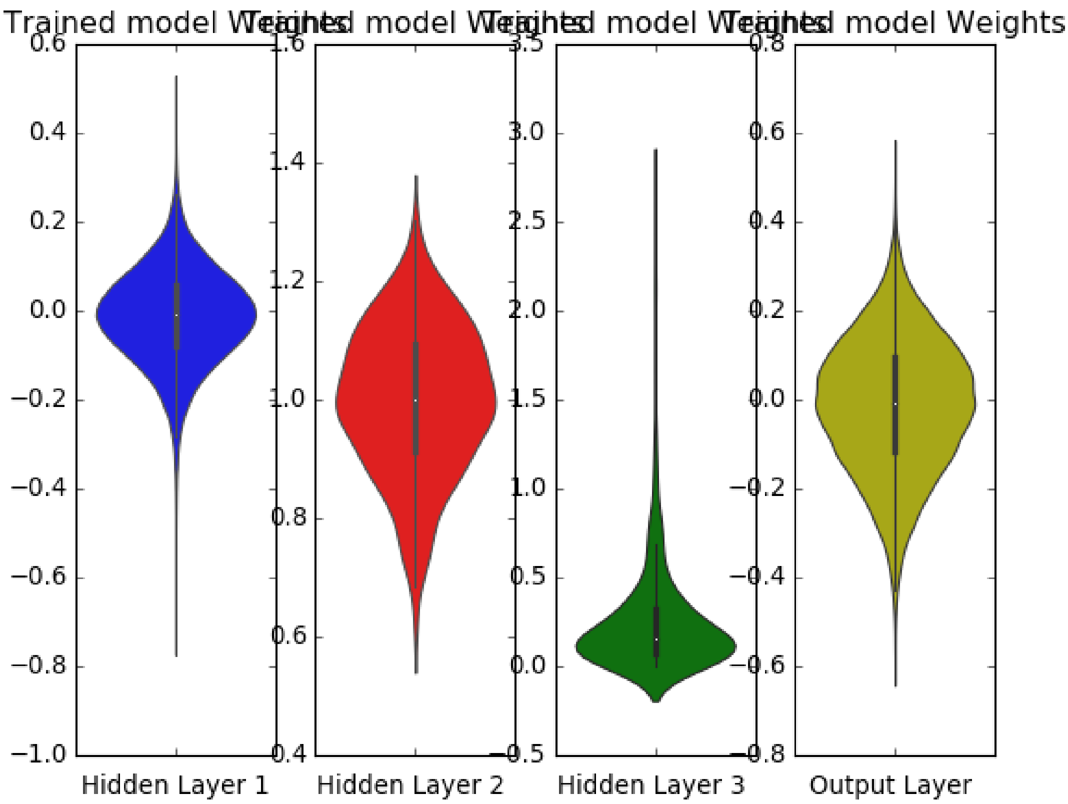
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.07038851246719714
Test accuracy: 0.9825000166893005
```



In [43]:

```
w_after = model_drop.get_weights()

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


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

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

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

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



MLP + ReLU + ADAM with 3 hidden layers with Dropout (dropout rate = 0.25)

In [48]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.25))

model_drop.add(Dense(128, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.25))

model_drop.add(Dense(52, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.25))

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


model_drop.summary()
```

```
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_39 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_23 (Batc (None, 364)               1456      
_________________________________________________________________
dropout_18 (Dropout)         (None, 364)               0         
_________________________________________________________________
dense_40 (Dense)             (None, 128)               46720     
_________________________________________________________________
batch_normalization_24 (Batc (None, 128)               512       
_________________________________________________________________
dropout_19 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_41 (Dense)             (None, 52)                6708      
_________________________________________________________________
batch_normalization_25 (Batc (None, 52)                208       
_________________________________________________________________
dropout_20 (Dropout)         (None, 52)                0         
_________________________________________________________________
dense_42 (Dense)             (None, 10)                530       
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

In [49]:

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

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

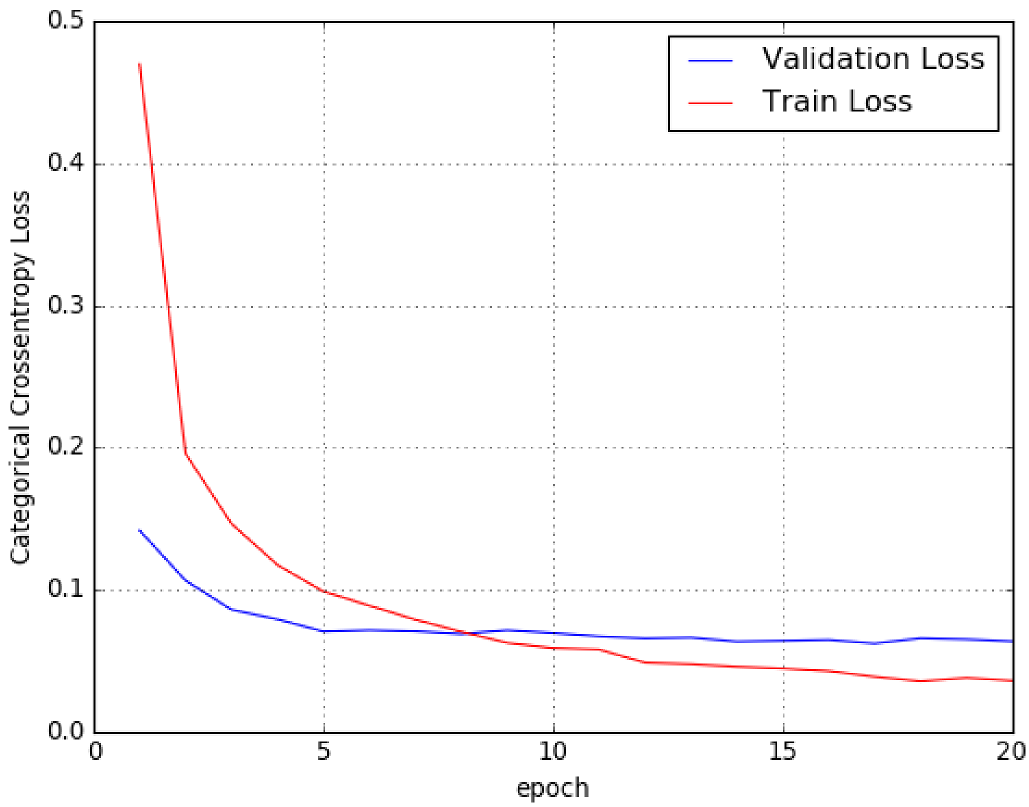
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.469
6 - accuracy: 0.8598 - val_loss: 0.1418 - val_accuracy: 0.9546
Epoch 2/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.195
8 - accuracy: 0.9419 - val_loss: 0.1066 - val_accuracy: 0.9661
Epoch 3/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.146
5 - accuracy: 0.9567 - val_loss: 0.0860 - val_accuracy: 0.9710
Epoch 4/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.117
5 - accuracy: 0.9639 - val_loss: 0.0793 - val_accuracy: 0.9748
Epoch 5/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.098
9 - accuracy: 0.9702 - val_loss: 0.0707 - val_accuracy: 0.9779
Epoch 6/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0889
- accuracy: 0.9737 - val_loss: 0.0717 - val_accuracy: 0.9786
Epoch 7/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.079
1 - accuracy: 0.9751 - val_loss: 0.0709 - val_accuracy: 0.9774
Epoch 8/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.070
6 - accuracy: 0.9783 - val_loss: 0.0690 - val_accuracy: 0.9801
Epoch 9/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.062
8 - accuracy: 0.9807 - val_loss: 0.0716 - val_accuracy: 0.9802
Epoch 10/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0590
- accuracy: 0.9816 - val_loss: 0.0696 - val_accuracy: 0.9800
Epoch 11/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.058
1 - accuracy: 0.9820 - val_loss: 0.0673 - val_accuracy: 0.9813
Epoch 12/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.048
8 - accuracy: 0.9846 - val_loss: 0.0659 - val_accuracy: 0.9811
Epoch 13/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0477
- accuracy: 0.9854 - val_loss: 0.0665 - val_accuracy: 0.9817
Epoch 14/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0460
- accuracy: 0.9852 - val_loss: 0.0636 - val_accuracy: 0.9830
Epoch 15/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0448
- accuracy: 0.9861 - val_loss: 0.0642 - val_accuracy: 0.9833
Epoch 16/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.042
9 - accuracy: 0.9863 - val_loss: 0.0648 - val_accuracy: 0.9816
Epoch 17/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.038
9 - accuracy: 0.9871 - val_loss: 0.0623 - val_accuracy: 0.9831
Epoch 18/20
```

```
60000/60000 [==============================] - 6s 99us/step - loss: 0.0358
- accuracy: 0.9888 - val_loss: 0.0660 - val_accuracy: 0.9828
Epoch 19/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0381
- accuracy: 0.9876 - val_loss: 0.0652 - val_accuracy: 0.9833
Epoch 20/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0362
- accuracy: 0.9885 - val_loss: 0.0638 - val_accuracy: 0.9820
```

In [50]:

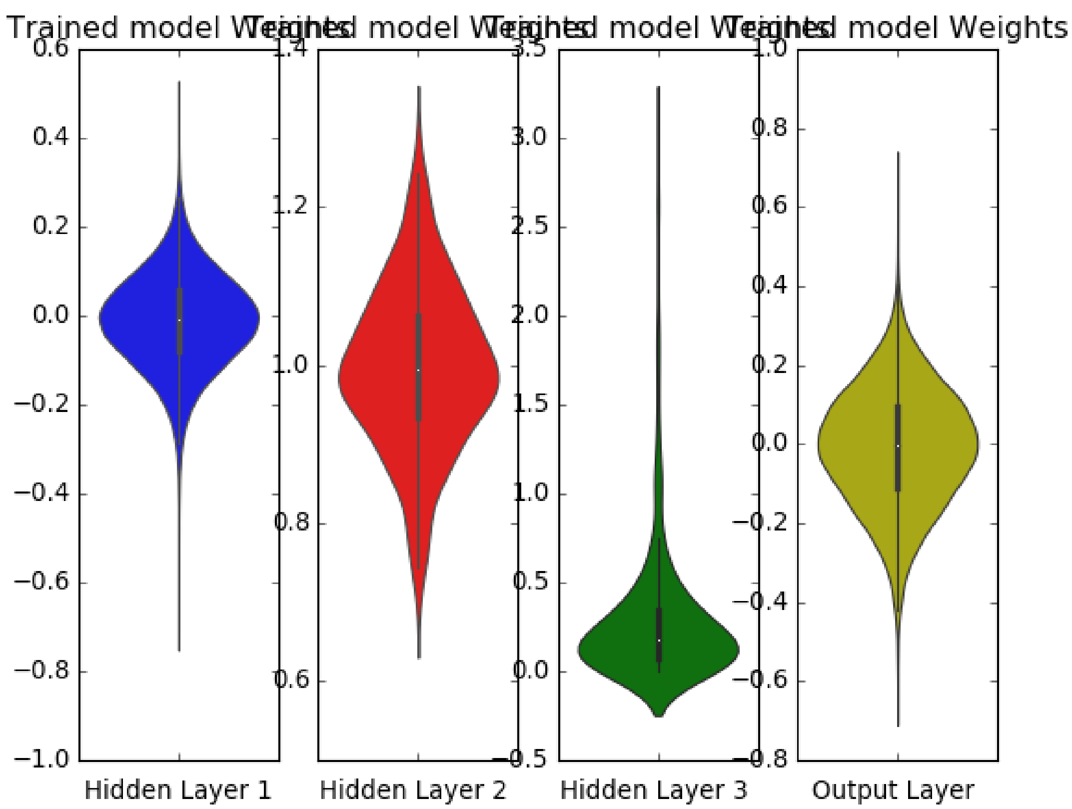
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.06377539714767481
Test accuracy: 0.9819999933242798
```



In [51]:

```
w_after = model_drop.get_weights()

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


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

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

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

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



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

In [52]:

```
model_relu = Sequential()
model_relu.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdd
model_relu.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

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

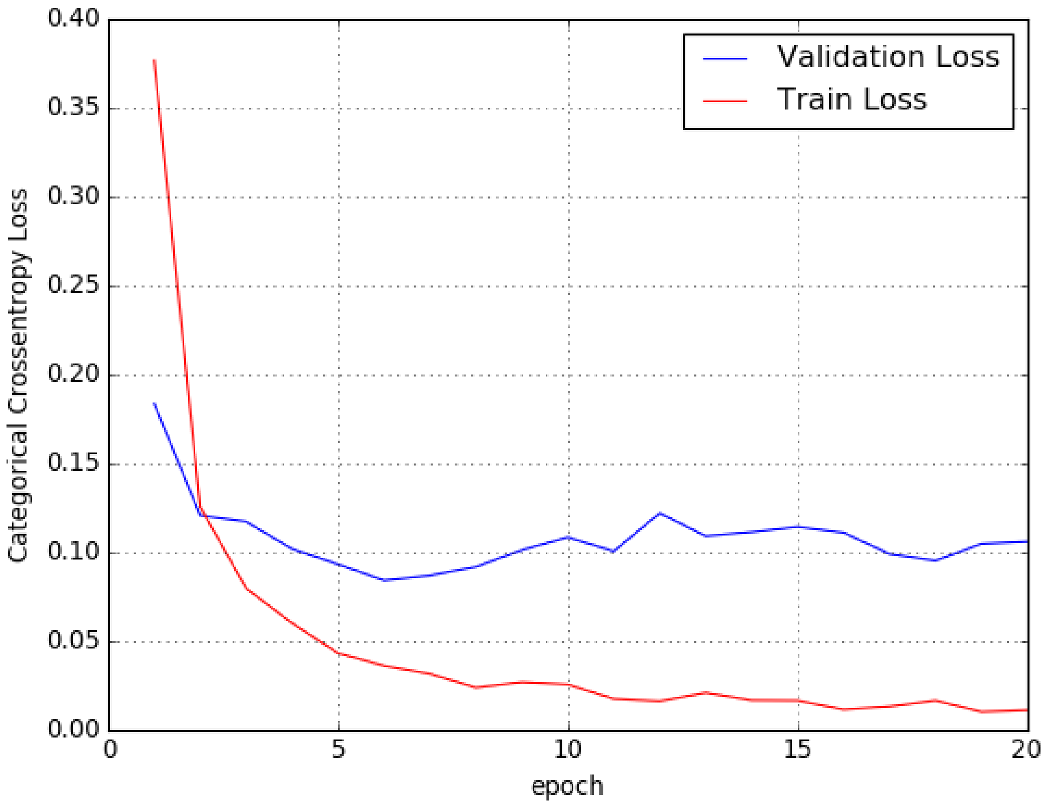
```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_43 (Dense)             (None, 364)               285740    
_________________________________________________________________
dense_44 (Dense)             (None, 128)               46720     
_________________________________________________________________
dense_45 (Dense)             (None, 64)                8256      
_________________________________________________________________
dense_46 (Dense)             (None, 32)                2080      
_________________________________________________________________
dense_47 (Dense)             (None, 16)                528       
_________________________________________________________________
dense_48 (Dense)             (None, 10)                170       
=================================================================
Total params: 343,494
Trainable params: 343,494
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.3766 -
accuracy: 0.8903 - val_loss: 0.1837 - val_accuracy: 0.9487
Epoch 2/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.1257 -
accuracy: 0.9641 - val_loss: 0.1208 - val_accuracy: 0.9634
Epoch 3/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0797 -
accuracy: 0.9764 - val_loss: 0.1173 - val_accuracy: 0.9648
Epoch 4/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0601 -
accuracy: 0.9812 - val_loss: 0.1019 - val_accuracy: 0.9697
Epoch 5/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0433 -
accuracy: 0.9869 - val_loss: 0.0933 - val_accuracy: 0.9751
Epoch 6/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0362 -
accuracy: 0.9883 - val_loss: 0.0842 - val_accuracy: 0.9771
Epoch 7/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0316 -
accuracy: 0.9899 - val_loss: 0.0870 - val_accuracy: 0.9763
Epoch 8/20
```

```
60000/60000 [==============================] - 4s 65us/step - loss: 0.0239 -
accuracy: 0.9920 - val_loss: 0.0919 - val_accuracy: 0.9753
Epoch 9/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0269 -
accuracy: 0.9913 - val_loss: 0.1013 - val_accuracy: 0.9760
Epoch 10/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0257 -
accuracy: 0.9910 - val_loss: 0.1084 - val_accuracy: 0.9755
Epoch 11/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0176 -
accuracy: 0.9947 - val_loss: 0.1006 - val_accuracy: 0.9777
Epoch 12/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0163 -
accuracy: 0.9947 - val_loss: 0.1221 - val_accuracy: 0.9752
Epoch 13/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0210 -
accuracy: 0.9934 - val_loss: 0.1091 - val_accuracy: 0.9754
Epoch 14/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0168 -
accuracy: 0.9944 - val_loss: 0.1114 - val_accuracy: 0.9762
Epoch 15/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0166 -
accuracy: 0.9946 - val_loss: 0.1143 - val_accuracy: 0.9749
Epoch 16/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0116 -
accuracy: 0.9962 - val_loss: 0.1110 - val_accuracy: 0.9798
Epoch 17/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0133 -
accuracy: 0.9959 - val_loss: 0.0991 - val_accuracy: 0.9784
Epoch 18/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.0166 -
accuracy: 0.9947 - val_loss: 0.0953 - val_accuracy: 0.9801
Epoch 19/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0103 -
accuracy: 0.9968 - val_loss: 0.1048 - val_accuracy: 0.9776
Epoch 20/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0112 -
accuracy: 0.9966 - val_loss: 0.1061 - val_accuracy: 0.9777
```

In [53]:

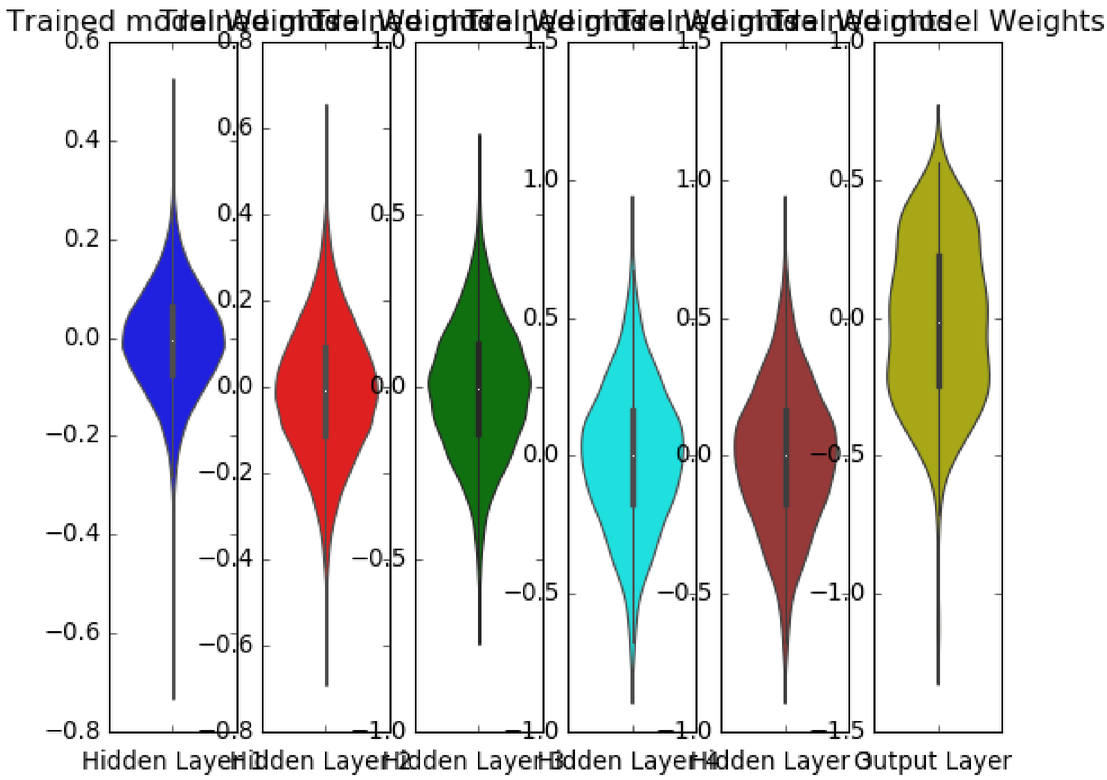
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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


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

```
Test score: 0.1060613916828221
Test accuracy: 0.9776999950408936
```



In [54]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

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


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



MLP + ReLU + ADAM with 5 hidden layers with Batch Normalisation

In [55]:

```
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(364, activation='relu', input_shape=(input_dim,),
                      kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_batch.add(BatchNormalization())

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

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

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

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

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


model_batch.summary()
```

Model: "sequential_14"

Layer (type)	Output Shape	Param #
dense_49 (Dense)	(None, 364)	285740
batch_normalization_26 (Batc	(None, 364)	1456
dense_50 (Dense)	(None, 128)	46720
batch_normalization_27 (Batc	(None, 128)	512
dense_51 (Dense)	(None, 64)	8256
batch_normalization_28 (Batc	(None, 64)	256
dense_52 (Dense)	(None, 32)	2080
batch_normalization_29 (Batc	(None, 32)	128
dense_53 (Dense)	(None, 16)	528
batch_normalization_30 (Batc	(None, 16)	64
dense_54 (Dense)	(None, 10)	170

```
=================================================================
Total params: 345,910
Trainable params: 344,702
Non-trainable params: 1,208
_________________________________________________________________
```

In [56]:

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

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

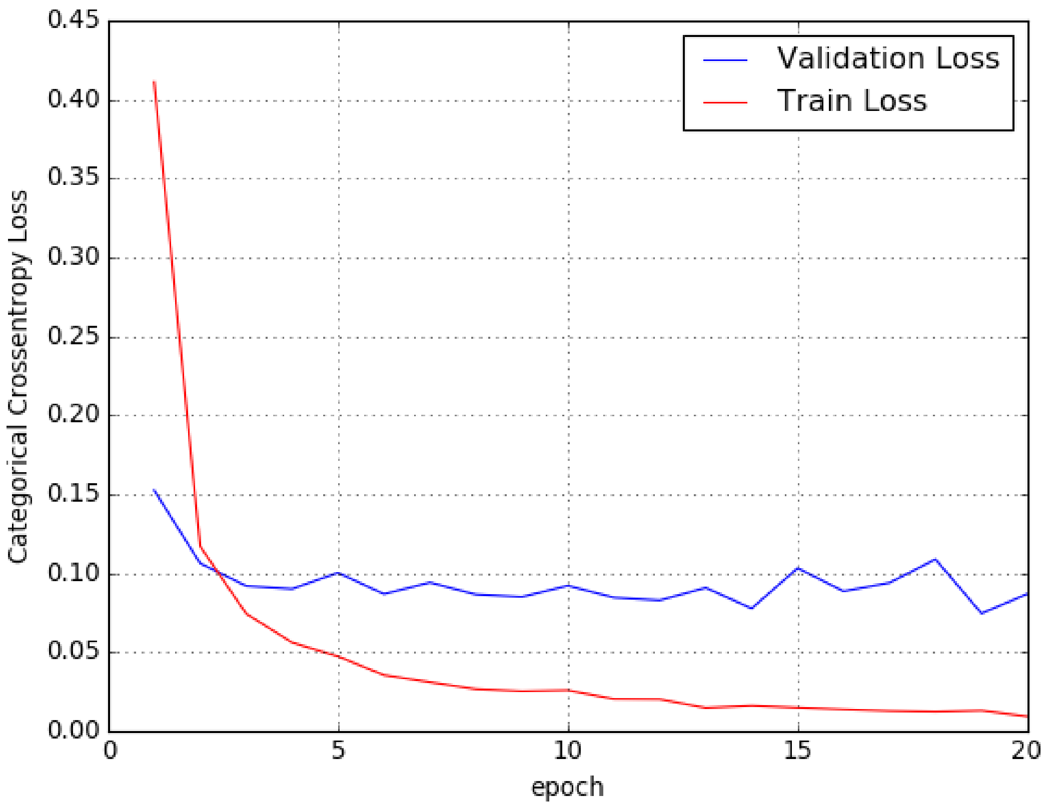
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.4110
- accuracy: 0.8947 - val_loss: 0.1524 - val_accuracy: 0.9563
Epoch 2/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.1170
- accuracy: 0.9677 - val_loss: 0.1063 - val_accuracy: 0.9708
Epoch 3/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0744 -
accuracy: 0.9780 - val_loss: 0.0919 - val_accuracy: 0.9743
Epoch 4/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0561 -
accuracy: 0.9832 - val_loss: 0.0901 - val_accuracy: 0.9741
Epoch 5/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0473 -
accuracy: 0.9853 - val_loss: 0.1002 - val_accuracy: 0.9725
Epoch 6/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0354
- accuracy: 0.9887 - val_loss: 0.0868 - val_accuracy: 0.9762
Epoch 7/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0310 -
accuracy: 0.9905 - val_loss: 0.0941 - val_accuracy: 0.9741
Epoch 8/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0266 -
accuracy: 0.9915 - val_loss: 0.0865 - val_accuracy: 0.9769
Epoch 9/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0252 -
accuracy: 0.9916 - val_loss: 0.0849 - val_accuracy: 0.9761
Epoch 10/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0258 -
accuracy: 0.9917 - val_loss: 0.0921 - val_accuracy: 0.9756
Epoch 11/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0204
- accuracy: 0.9933 - val_loss: 0.0846 - val_accuracy: 0.9771
Epoch 12/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0202
- accuracy: 0.9936 - val_loss: 0.0828 - val_accuracy: 0.9783
Epoch 13/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0147 -
accuracy: 0.9951 - val_loss: 0.0908 - val_accuracy: 0.9764
Epoch 14/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0161 -
accuracy: 0.9946 - val_loss: 0.0777 - val_accuracy: 0.9807
Epoch 15/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0148 -
accuracy: 0.9949 - val_loss: 0.1032 - val_accuracy: 0.9747
Epoch 16/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0137
- accuracy: 0.9956 - val_loss: 0.0886 - val_accuracy: 0.9780
Epoch 17/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0128
- accuracy: 0.9959 - val_loss: 0.0939 - val_accuracy: 0.9780
Epoch 18/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0123
```

```
- accuracy: 0.9959 - val_loss: 0.1088 - val_accuracy: 0.9741
Epoch 19/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0130 -
accuracy: 0.9956 - val_loss: 0.0745 - val_accuracy: 0.9827
Epoch 20/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0093
- accuracy: 0.9970 - val_loss: 0.0868 - val_accuracy: 0.9798
```

In [57]:

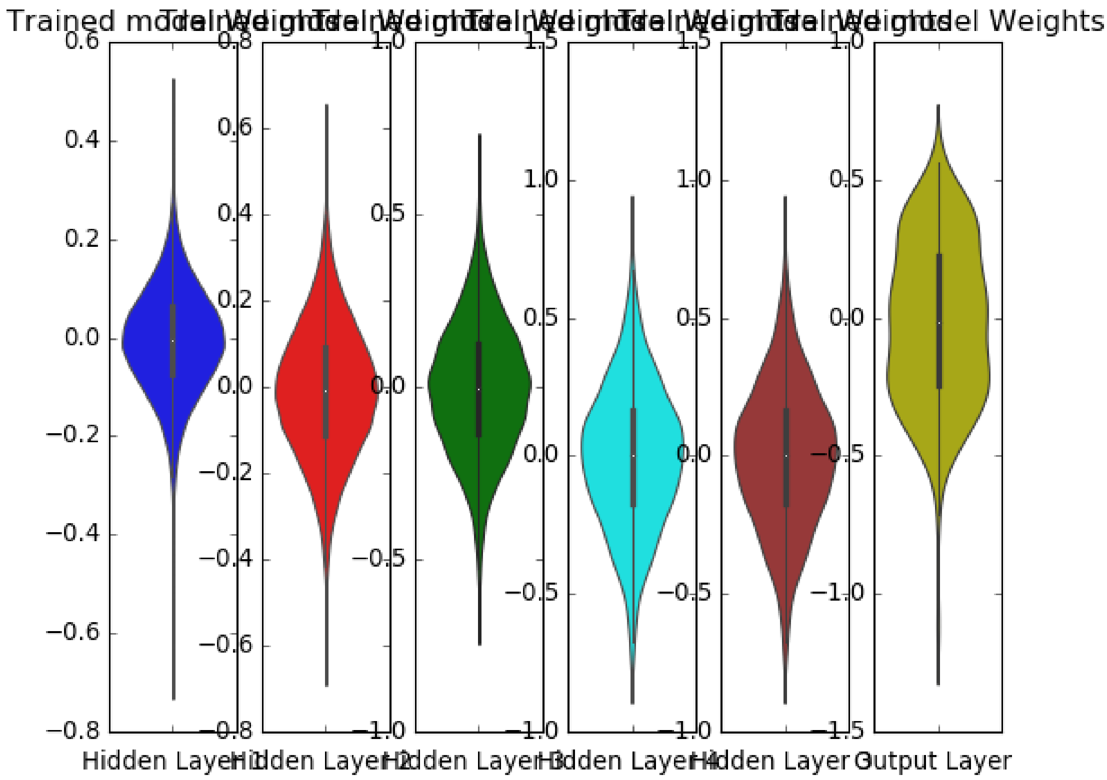
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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


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

```
Test score: 0.1060613916828221
Test accuracy: 0.9776999950408936
```



In [58]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

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


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



MLP + ReLU + ADAM with 5 hidden layers with Dropout (dropout rate = 0.5)

In [59]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_drop.add(Dropout(0.5))

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

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

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

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

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


model_drop.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_55 (Dense)             (None, 364)               285740    
_________________________________________________________________
dropout_21 (Dropout)         (None, 364)               0         
_________________________________________________________________
dense_56 (Dense)             (None, 128)               46720     
_________________________________________________________________
dropout_22 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_57 (Dense)             (None, 64)                8256      
_________________________________________________________________
dropout_23 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_58 (Dense)             (None, 32)                2080      
_________________________________________________________________
dropout_24 (Dropout)         (None, 32)                0         
_________________________________________________________________
dense_59 (Dense)             (None, 16)                528       
_________________________________________________________________
dropout_25 (Dropout)         (None, 16)                0         
_________________________________________________________________
dense_60 (Dense)             (None, 10)                170       
=================================================================
Total params: 343,494
Trainable params: 343,494
Non-trainable params: 0
_________________________________________________________________
```

In [60]:

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

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

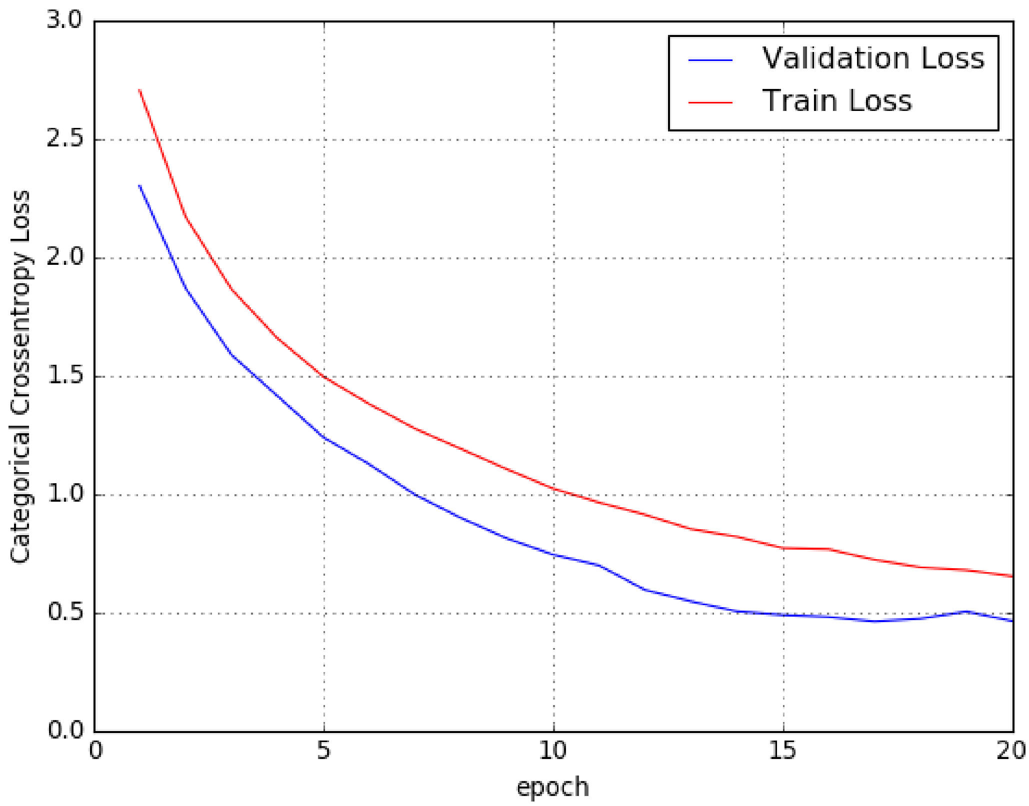
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 98us/step - loss: 2.7044 -
accuracy: 0.1195 - val_loss: 2.3014 - val_accuracy: 0.1136
Epoch 2/20
60000/60000 [==============================] - 5s 83us/step - loss: 2.1709 -
accuracy: 0.2078 - val_loss: 1.8698 - val_accuracy: 0.3667
Epoch 3/20
60000/60000 [==============================] - 5s 80us/step - loss: 1.8653 -
accuracy: 0.3251 - val_loss: 1.5882 - val_accuracy: 0.4043
Epoch 4/20
60000/60000 [==============================] - 5s 80us/step - loss: 1.6584 -
accuracy: 0.4065 - val_loss: 1.4147 - val_accuracy: 0.5350
Epoch 5/20
60000/60000 [==============================] - 5s 80us/step - loss: 1.4961 -
accuracy: 0.4598 - val_loss: 1.2402 - val_accuracy: 0.5695
Epoch 6/20
60000/60000 [==============================] - 5s 81us/step - loss: 1.3807 -
accuracy: 0.4925 - val_loss: 1.1266 - val_accuracy: 0.6056
Epoch 7/20
60000/60000 [==============================] - 5s 83us/step - loss: 1.2763 -
accuracy: 0.5304 - val_loss: 0.9972 - val_accuracy: 0.6800
Epoch 8/20
60000/60000 [==============================] - 5s 81us/step - loss: 1.1909 -
accuracy: 0.5656 - val_loss: 0.8996 - val_accuracy: 0.7056
Epoch 9/20
60000/60000 [==============================] - 5s 81us/step - loss: 1.1053 -
accuracy: 0.6075 - val_loss: 0.8130 - val_accuracy: 0.7336
Epoch 10/20
60000/60000 [==============================] - 6s 93us/step - loss: 1.0238 -
accuracy: 0.6384 - val_loss: 0.7449 - val_accuracy: 0.7680
Epoch 11/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.9651 -
accuracy: 0.6621 - val_loss: 0.6997 - val_accuracy: 0.8079
Epoch 12/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.9139 -
accuracy: 0.6915 - val_loss: 0.5961 - val_accuracy: 0.8414
Epoch 13/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.8529 -
accuracy: 0.7160 - val_loss: 0.5478 - val_accuracy: 0.8608
Epoch 14/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.8207 -
accuracy: 0.7338 - val_loss: 0.5052 - val_accuracy: 0.8943
Epoch 15/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.7725 -
accuracy: 0.7565 - val_loss: 0.4904 - val_accuracy: 0.8933
Epoch 16/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.7675 -
accuracy: 0.7672 - val_loss: 0.4819 - val_accuracy: 0.9102
Epoch 17/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.7231 -
accuracy: 0.7794 - val_loss: 0.4629 - val_accuracy: 0.9135
Epoch 18/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.6916 -
```

```
accuracy: 0.7927 - val_loss: 0.4751 - val_accuracy: 0.9215
Epoch 19/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.6795 -
accuracy: 0.7985 - val_loss: 0.5045 - val_accuracy: 0.9202
Epoch 20/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.6536 -
accuracy: 0.8087 - val_loss: 0.4650 - val_accuracy: 0.9332
```

In [61]:

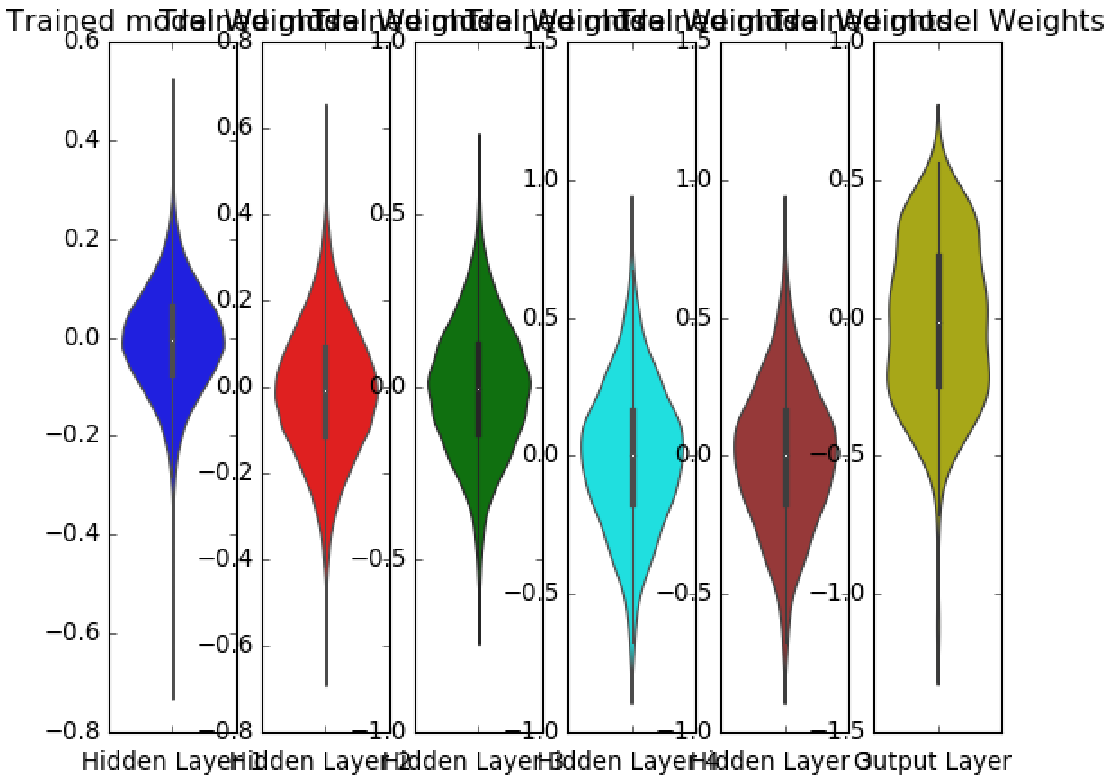
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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


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

```
Test score: 0.1060613916828221
Test accuracy: 0.9776999950408936
```



In [62]:

```
w_after = model_relu.get_weights()

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


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

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

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

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

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


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



MLP + ReLU + ADAM with 5 hidden layers with Dropout (dropout rate = 0.25)

In [63]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(364, activation='relu', input_shape=(input_dim,),
                     kernel_initializer=RandomNormal(mean=0.0, stddev=0.074, seed=None)))
model_drop.add(Dropout(0.25))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdc
model_drop.add(Dropout(0.25))

model_drop.add(Dense(64, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.25))

model_drop.add(Dense(32, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.25))

model_drop.add(Dense(16, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stdde
model_drop.add(Dropout(0.25))

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


model_drop.summary()
```

```
Model: "sequential_16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_61 (Dense)             (None, 364)               285740    
_________________________________________________________________
dropout_26 (Dropout)         (None, 364)               0         
_________________________________________________________________
dense_62 (Dense)             (None, 128)               46720     
_________________________________________________________________
dropout_27 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_63 (Dense)             (None, 64)                8256      
_________________________________________________________________
dropout_28 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_64 (Dense)             (None, 32)                2080      
_________________________________________________________________
dropout_29 (Dropout)         (None, 32)                0         
_________________________________________________________________
dense_65 (Dense)             (None, 16)                528       
_________________________________________________________________
dropout_30 (Dropout)         (None, 16)                0         
_________________________________________________________________
dense_66 (Dense)             (None, 10)                170       
=================================================================
Total params: 343,494
Trainable params: 343,494
Non-trainable params: 0
_________________________________________________________________
```

In [64]:

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

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

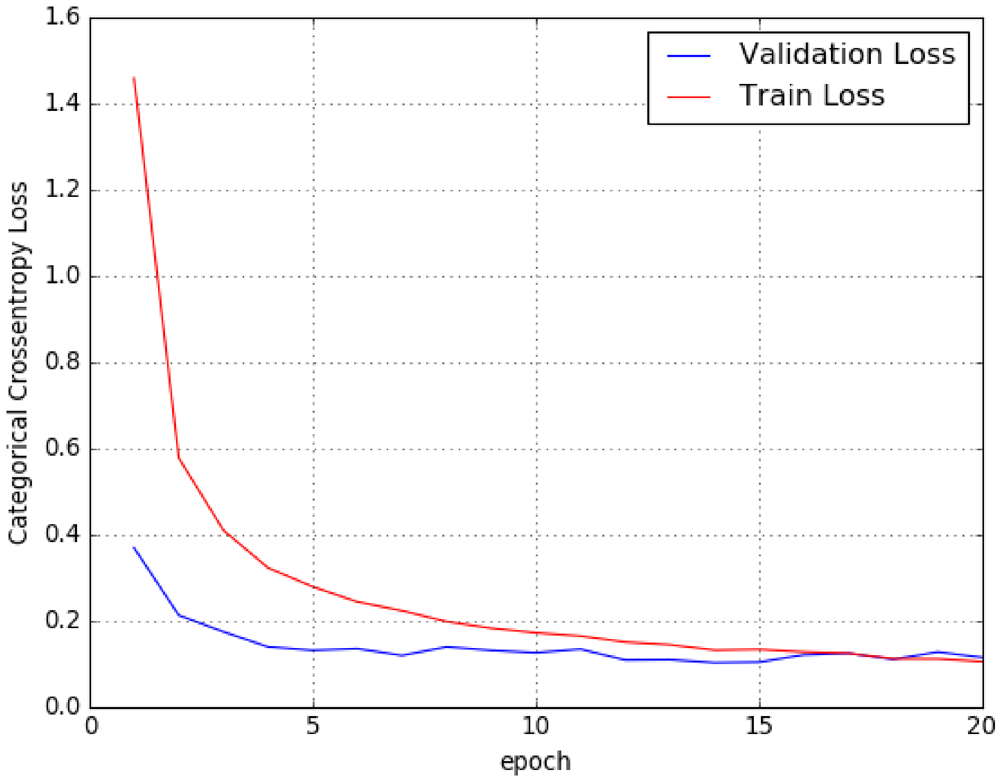
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 105us/step - loss: 1.4585
- accuracy: 0.5030 - val_loss: 0.3698 - val_accuracy: 0.9152
Epoch 2/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.5783 -
accuracy: 0.8296 - val_loss: 0.2133 - val_accuracy: 0.9456
Epoch 3/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.4104 -
accuracy: 0.8878 - val_loss: 0.1751 - val_accuracy: 0.9563
Epoch 4/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.3233 -
accuracy: 0.9148 - val_loss: 0.1400 - val_accuracy: 0.9671
Epoch 5/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.2796 -
accuracy: 0.9281 - val_loss: 0.1321 - val_accuracy: 0.9671
Epoch 6/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.2445 -
accuracy: 0.9377 - val_loss: 0.1359 - val_accuracy: 0.9683
Epoch 7/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.2239 -
accuracy: 0.9448 - val_loss: 0.1203 - val_accuracy: 0.9719
Epoch 8/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.1987 -
accuracy: 0.9500 - val_loss: 0.1399 - val_accuracy: 0.9725
Epoch 9/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.1833 -
accuracy: 0.9551 - val_loss: 0.1320 - val_accuracy: 0.9730
Epoch 10/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.1727 -
accuracy: 0.9566 - val_loss: 0.1266 - val_accuracy: 0.9747
Epoch 11/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.1652 -
accuracy: 0.9588 - val_loss: 0.1348 - val_accuracy: 0.9733
Epoch 12/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.1513 -
accuracy: 0.9623 - val_loss: 0.1096 - val_accuracy: 0.9768
Epoch 13/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.1451 -
accuracy: 0.9639 - val_loss: 0.1105 - val_accuracy: 0.9778
Epoch 14/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.1326 -
accuracy: 0.9662 - val_loss: 0.1034 - val_accuracy: 0.9805
Epoch 15/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.1343 -
accuracy: 0.9657 - val_loss: 0.1047 - val_accuracy: 0.9790
Epoch 16/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.1284 -
accuracy: 0.9679 - val_loss: 0.1215 - val_accuracy: 0.9792
Epoch 17/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.1253 -
accuracy: 0.9683 - val_loss: 0.1252 - val_accuracy: 0.9801
Epoch 18/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.1126 -
```

```
accuracy: 0.9715 - val_loss: 0.1112 - val_accuracy: 0.9800
Epoch 19/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.1124 -
accuracy: 0.9712 - val_loss: 0.1283 - val_accuracy: 0.9788
Epoch 20/20
60000/60000 [==============================] - 5s 80us/step - loss: 0.1062 -
accuracy: 0.9738 - val_loss: 0.1156 - val_accuracy: 0.9803
```

In [65]:

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

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

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

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

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

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


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

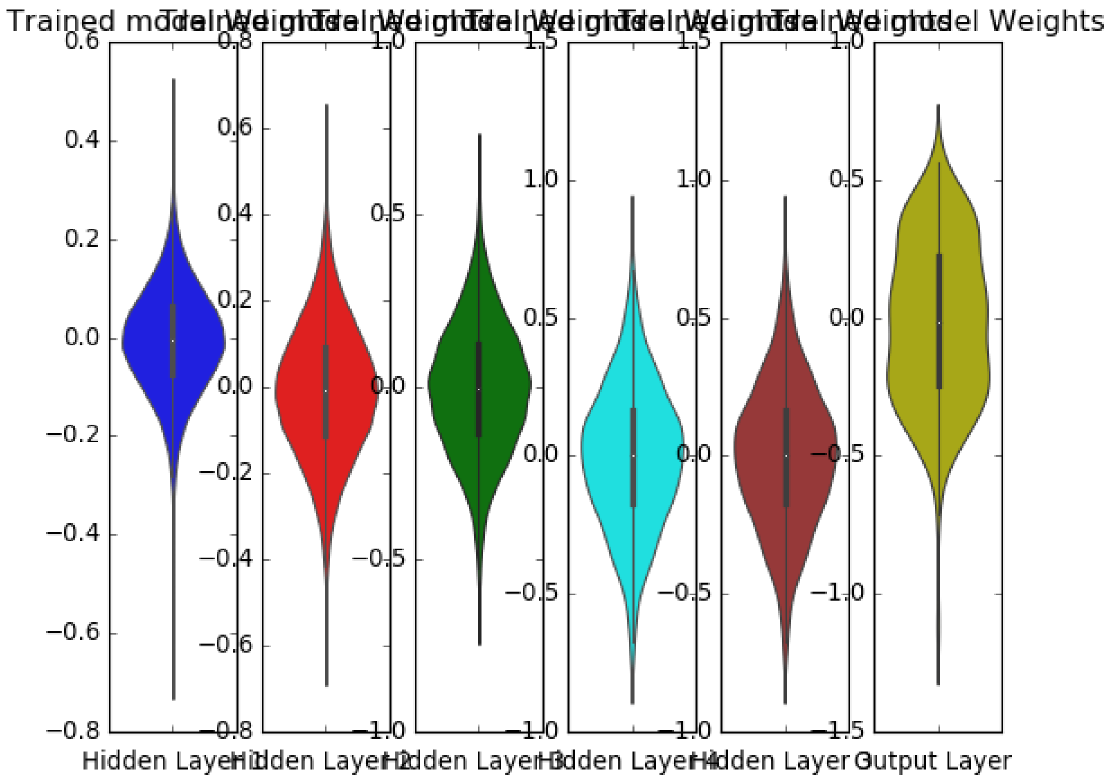
```
Test score: 0.1060613916828221
Test accuracy: 0.9776999950408936

/home/komalumrethe/anaconda3/lib/python3.5/site-packages/matplotlib/pyplot.p
y:524: RuntimeWarning: More than 20 figures have been opened. Figures create
d through the pyplot interface (`matplotlib.pyplot.figure`) are retained unt
il explicitly closed and may consume too much memory. (To control this warni
ng, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```


Validation Loss
Train Loss
Categorical Crossentropy Loss
epoch
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
5
10
15
20

In [66]:

```
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()
```

```
/home/komalumrethe/anaconda3/lib/python3.5/site-packages/matplotlib/pyplot.p
y:524: RuntimeWarning: More than 20 figures have been opened. Figures create
d through the pyplot interface (`matplotlib.pyplot.figure`) are retained unt
il explicitly closed and may consume too much memory. (To control this warni
ng, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```


Trained model Weights
Hidden Layer 1
Hidden Layer 2
Hidden Layer 3
Hidden Layer 4
Hidden Layer 5
Output Layer

In [67]:

```
from prettytable import PrettyTable

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

x.add_row(["2 layer", "Without Batch Normalization and Dropout", 97.92])
x.add_row(["2 layer", "With Batch Normalization", 97.96])
x.add_row(["2 layer", "With Dropuot(dropout rate = 0.5)", 98.28])
x.add_row(["2 layer", "With Dropuot(dropout rate = 0.25)", 98.29])
x.add_row(["2 layer", "With Dropuot(dropout rate = 0.75)", 98.35])

x.add_row(["3 layer", "Without Batch Normalization and Dropout", 97.90])
x.add_row(["3 layer", "With Batch Normalization", 97.89])
x.add_row(["3 layer", "With Dropuot(dropout rate = 0.5)", 98.25])
x.add_row(["3 layer", "With Dropuot(dropout rate = 0.25)", 98.19])

x.add_row(["5 layer", "Without Batch Normalization and Dropout", 97.7699])
x.add_row(["5 layer", "With Batch Normalization", 97.7699])
x.add_row(["5 layer", "With Dropuot(dropout rate = 0.5)", 97.7699])
x.add_row(["5 layer", "With Dropuot(dropout rate = 0.25)", 97.7699])

print(x)
```

```
+--------------+-----------------------------------------+----------+
| Architecture |                parameters               | Accuracy |
+--------------+-----------------------------------------+----------+
|   2 layer    | Without Batch Normalization and Dropout |  97.92   |
|   2 layer    |         With Batch Normalization        |  97.96   |
|   2 layer    |     With Dropuot(dropout rate = 0.5)    |  98.28   |
|   2 layer    |    With Dropuot(dropout rate = 0.25)    |  98.29   |
|   2 layer    |    With Dropuot(dropout rate = 0.75)    |  98.35   |
|   3 layer    | Without Batch Normalization and Dropout |   97.9   |
|   3 layer    |         With Batch Normalization        |  97.89   |
|   3 layer    |     With Dropuot(dropout rate = 0.5)    |  98.25   |
|   3 layer    |    With Dropuot(dropout rate = 0.25)    |  98.19   |
|   5 layer    | Without Batch Normalization and Dropout | 97.7699  |
|   5 layer    |         With Batch Normalization        | 97.7699  |
|   5 layer    |     With Dropuot(dropout rate = 0.5)    | 97.7699  |
|   5 layer    |    With Dropuot(dropout rate = 0.25)    | 97.7699  |
+--------------+-----------------------------------------+----------+
```

Procedure Followed

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