In [1]:

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

```
Using TensorFlow backend.
```

In [2]:

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
import time
# https://gist.github.com/greydanus/f6eee59eaf1d90fcb3b534a25362cea4
# https://stackoverflow.com/a/14434334
# this function is used to update the plots for each epoch and error
def plt_dynamic(x, vy, ty, ax, colors=['b']):
    ax.plot(x, vy, 'b', label="Validation Loss")
    ax.plot(x, ty, 'r', label="Train Loss")
    plt.legend()
    plt.grid()
    fig.canvas.draw()
```

In [3]:

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

In [4]:

```
print("Number of training examples :", X_train.shape[0], "and each image is of shape (%d, %d)"%(X_
train.shape[1], X_train.shape[2]))
print("Number of training examples :", X_test.shape[0], "and each image is of shape (%d,
%d)"%(X_test.shape[1], X_test.shape[2]))
```

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

In [5]:

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

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

In [6]:

```
# after converting the input images from 3d to 2d vectors

print("Number of training examples :", X_train.shape[0], "and each image is of shape
(%d)"%(X_train.shape[1]))
print("Number of training examples :", X_test.shape[0], "and each image is of shape (%d)"%(X_test.
shape[1]))
```

```
Number of training examples : 60000 and each image is of shape (784)
Number of training examples : 10000 and each image is of shape (784)
```

In [7]:

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

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

In [8]:

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

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

In [9]:

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

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
```

```
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.01176471 0.07058824 0.07058824 0.07058824
0.49411765 0.53333333 0.68627451 0.10196078 0.65098039 1.
0.96862745 0.49803922 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.11764706 0.14117647 0.36862745 0.60392157
0.66666667 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
0.88235294 0.6745098  0.99215686 0.94901961 0.76470588 0.25098039
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.19215686
0.93333333 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
0.99215686 0.99215686 0.99215686 0.98431373 0.36470588 0.32156863
0.32156863 0.21960784 0.15294118 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.07058824 0.85882353 0.99215686
0.99215686 0.99215686 0.99215686 0.99215686 0.77647059 0.71372549
0.96862745 0.94509804 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.31372549 0.61176471 0.41960784 0.99215686
0.99215686 0.80392157 0.04313725 0.         0.16862745 0.60392157
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.05490196 0.00392157 0.60392157 0.99215686 0.35294118
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.54509804 0.99215686 0.74509804 0.00784314 0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.04313725
0.74509804 0.99215686 0.2745098  0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.1372549  0.94509804
0.88235294 0.62745098 0.42352941 0.00392157 0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.31764706 0.94117647 0.99215686
0.99215686 0.46666667 0.09803922 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.17647059 0.72941176 0.99215686 0.99215686
0.58823529 0.10588235 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.0627451  0.36470588 0.98823529 0.99215686 0.73333333
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.97647059 0.99215686 0.97647059 0.25098039 0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.18039216 0.50980392 0.71764706 0.99215686
0.99215686 0.81176471 0.00784314 0.         0.         0.
0.         0.         0.         0.         0.         0.
```

```
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.15294118 0.58039216
 0.89803922 0.99215686 0.99215686 0.99215686 0.98039216 0.71372549
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.09411765 0.44705882 0.86666667 0.99215686 0.99215686 0.99215686
 0.99215686 0.78823529 0.30588235 0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.09019608 0.25882353 0.83529412 0.99215686
 0.99215686 0.99215686 0.99215686 0.77647059 0.31764706 0.00784314
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.07058824 0.67058824
 0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.76470588
 0.31372549 0.03529412 0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.21568627 0.6745098  0.88627451 0.99215686 0.99215686 0.99215686
 0.99215686 0.95686275 0.52156863 0.04313725 0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.53333333 0.99215686
 0.99215686 0.99215686 0.83137255 0.52941176 0.51764706 0.0627451
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.        ]
```

In [10]:

```
# here we are having a class number for each image
print("Class label of first image :", y_train[0])

# lets convert this into a 10 dimensional vector
# ex: consider an image is 5 convert it into 5 => [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
# this conversion needed for MLPs

Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)

print("After converting the output into a vector : ",Y_train[0])
```

```
Class label of first image : 5
After converting the output into a vector :  [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Softmax classifier

In [11]:

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

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

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

```
#     Dense(10),
#     Activation('softmax'),
# ])

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

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

###

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

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

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

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

####

# https://keras.io/activations/

# Activations can either be used through an Activation layer, or through the activation argument s
upported by all forward layers:

# from keras.layers import Activation, Dense

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

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

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


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

In [12]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [13]:

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

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

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

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

In [14]:

```python
# Before training a model, you need to configure the learning process, which is done via the compi
le method

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


# Note: when using the categorical_crossentropy loss, your targets should be in categorical format

# (e.g. if you have 10 classes, the target for each sample should be a 10-dimensional vector that
is all-zeros except
# for a 1 at the index corresponding to the class of the sample).

# that is why we converted out labels into vectors

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

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

# fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
validation_split=0.0,
# validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, step
s_per_epoch=None,
# validation_steps=None)

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

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

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

history = model.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, validation
_data=(X_test, Y_test))
```

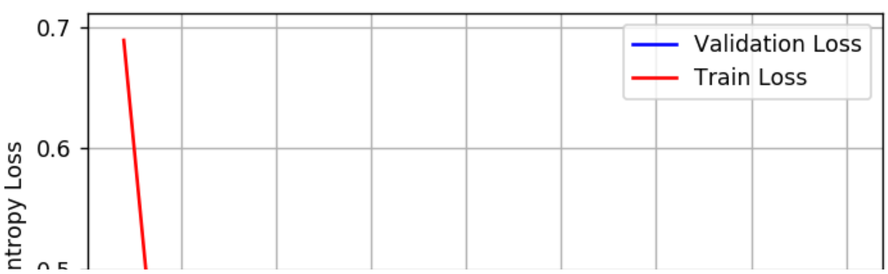
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 1s 22us/step - loss: 0.6891 - accuracy: 0.8268 - va
l_loss: 0.3872 - val_accuracy: 0.8995
Epoch 2/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.3632 - accuracy: 0.9024 - va
l_loss: 0.3214 - val_accuracy: 0.9117
Epoch 3/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.3197 - accuracy: 0.9123 - va
l_loss: 0.2960 - val_accuracy: 0.9178
Epoch 4/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2996 - accuracy: 0.9169 - va
l_loss: 0.2844 - val_accuracy: 0.9205
Epoch 5/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2879 - accuracy: 0.9200 - va
l_loss: 0.2771 - val_accuracy: 0.9237
Epoch 6/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.2796 - accuracy: 0.9218 - va
l_loss: 0.2752 - val_accuracy: 0.9224
Epoch 7/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.2735 - accuracy: 0.9237 - va
l_loss: 0.2716 - val_accuracy: 0.9238
Epoch 8/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2691 - accuracy: 0.9243 - va
l_loss: 0.2678 - val_accuracy: 0.9252
Epoch 9/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.2653 - accuracy: 0.9261 - va
l_loss: 0.2656 - val_accuracy: 0.9250
Epoch 10/20
60000/60000 [==============================] - 1s 15us/step - loss: 0.2622 - accuracy: 0.9265 - va
l_loss: 0.2659 - val_accuracy: 0.9250
Epoch 11/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2595 - accuracy: 0.9279 - va
```

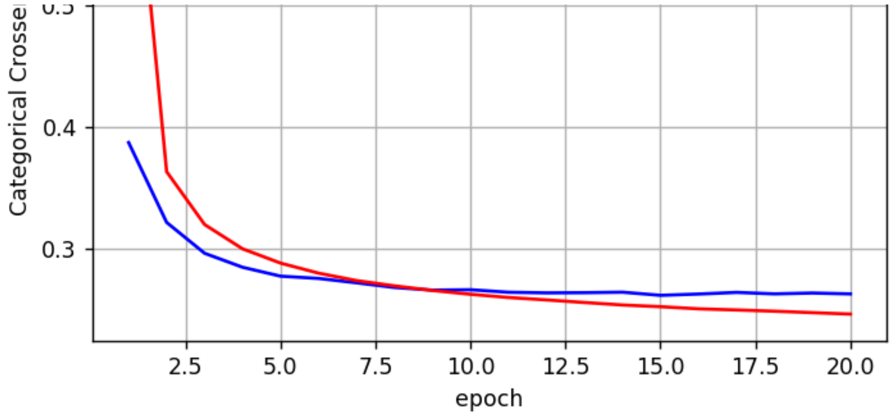
```
l_loss: 0.2639 - val_accuracy: 0.9255
Epoch 12/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.2575 - accuracy: 0.9284 - va
l_loss: 0.2634 - val_accuracy: 0.9255
Epoch 13/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2554 - accuracy: 0.9289 - va
l_loss: 0.2635 - val_accuracy: 0.9265
Epoch 14/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2534 - accuracy: 0.9297 - va
l_loss: 0.2639 - val_accuracy: 0.9259
Epoch 15/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2520 - accuracy: 0.9303 - va
l_loss: 0.2613 - val_accuracy: 0.9275
Epoch 16/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2502 - accuracy: 0.9313 - va
l_loss: 0.2624 - val_accuracy: 0.9260
Epoch 17/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2493 - accuracy: 0.9302 - va
l_loss: 0.2637 - val_accuracy: 0.9263
Epoch 18/20
60000/60000 [==============================] - 1s 16us/step - loss: 0.2482 - accuracy: 0.9312 - va
l_loss: 0.2625 - val_accuracy: 0.9280
Epoch 19/20
60000/60000 [==============================] - 1s 17us/step - loss: 0.2470 - accuracy: 0.9321 - va
l_loss: 0.2632 - val_accuracy: 0.9263
Epoch 20/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2459 - accuracy: 0.9322 - va
l_loss: 0.2625 - val_accuracy: 0.9268
```

In [15]:

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

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

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

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, va
lidation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.26249462166130544
Test accuracy: 0.926800012588501
```





ARCHITECTURE 1= 784--612--324--10(2- Hidden layers)

A) MLP+ReLu+ADAM

In [16]:

```
model_relu = Sequential()
model_relu.add(Dense(612, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(324, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125
, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, valid
ation_data=(X_test, Y_test))
```

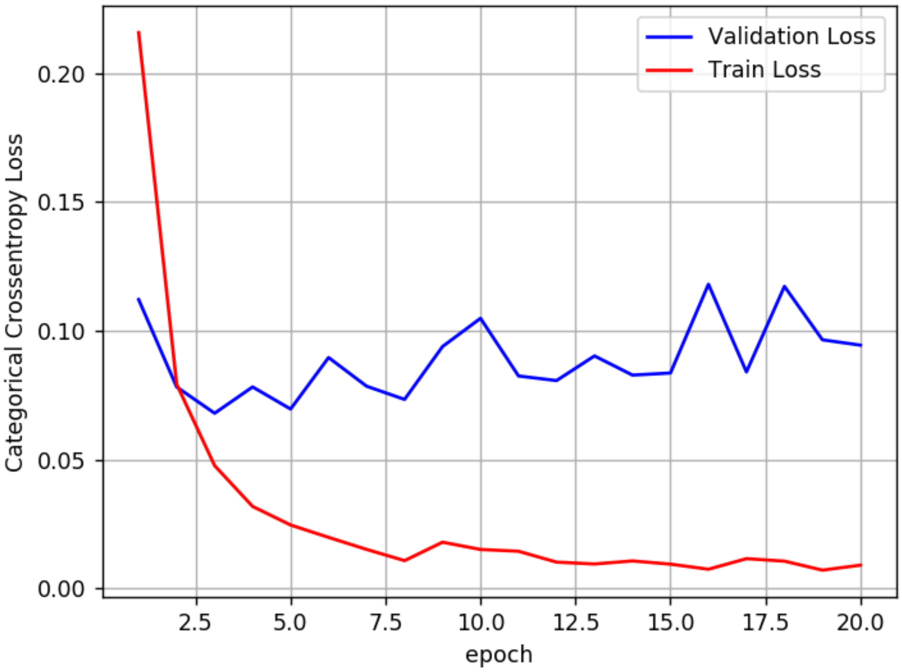
```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 612)               480420
_________________________________________________________________
dense_3 (Dense)              (None, 324)               198612
_________________________________________________________________
dense_4 (Dense)              (None, 10)                3250
=================================================================
Total params: 682,282
Trainable params: 682,282
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.2157 - accuracy: 0.9359 - va
l_loss: 0.1121 - val_accuracy: 0.9635
Epoch 2/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.0793 - accuracy: 0.9756 - va
l_loss: 0.0782 - val_accuracy: 0.9761
Epoch 3/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.0477 - accuracy: 0.9849 - va
l_loss: 0.0680 - val_accuracy: 0.9777
Epoch 4/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0318 - accuracy: 0.9902 - va
l_loss: 0.0782 - val_accuracy: 0.9758
Epoch 5/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.0246 - accuracy: 0.9918 - va
l_loss: 0.0696 - val_accuracy: 0.9778
Epoch 6/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.0198 - accuracy: 0.9935 - va
l_loss: 0.0896 - val_accuracy: 0.9748
Epoch 7/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0152 - accuracy: 0.9950 - va
```

```
l_loss: 0.0785 - val_accuracy: 0.9775
Epoch 8/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0108 - accuracy: 0.9967 - va
l_loss: 0.0734 - val_accuracy: 0.9814
Epoch 9/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.0180 - accuracy: 0.9940 - va
l_loss: 0.0939 - val_accuracy: 0.9755
Epoch 10/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0151 - accuracy: 0.9952 - va
l_loss: 0.1048 - val_accuracy: 0.9738
Epoch 11/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0144 - accuracy: 0.9952 - va
l_loss: 0.0825 - val_accuracy: 0.9802
Epoch 12/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0102 - accuracy: 0.9968 - va
l_loss: 0.0807 - val_accuracy: 0.9801
Epoch 13/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0095 - accuracy: 0.9965 - va
l_loss: 0.0902 - val_accuracy: 0.9805
Epoch 14/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0107 - accuracy: 0.9967 - va
l_loss: 0.0828 - val_accuracy: 0.9819
Epoch 15/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0094 - accuracy: 0.9969 - va
l_loss: 0.0836 - val_accuracy: 0.9826
Epoch 16/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.0075 - accuracy: 0.9977 - va
l_loss: 0.1181 - val_accuracy: 0.9770
Epoch 17/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.0116 - accuracy: 0.9962 - va
l_loss: 0.0840 - val_accuracy: 0.9826
Epoch 18/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0106 - accuracy: 0.9970 - va
l_loss: 0.1172 - val_accuracy: 0.9777
Epoch 19/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.0071 - accuracy: 0.9976 - va
l_loss: 0.0965 - val_accuracy: 0.9820
Epoch 20/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.0090 - accuracy: 0.9971 - va
l_loss: 0.0944 - val_accuracy: 0.9820
```

In [17]:

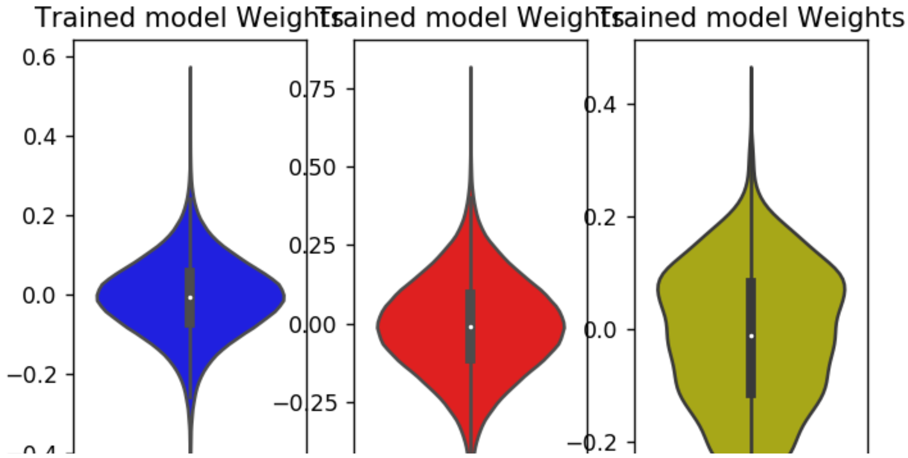
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, verbose=1, va
lidation_data=(X_test, Y_test))

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

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


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

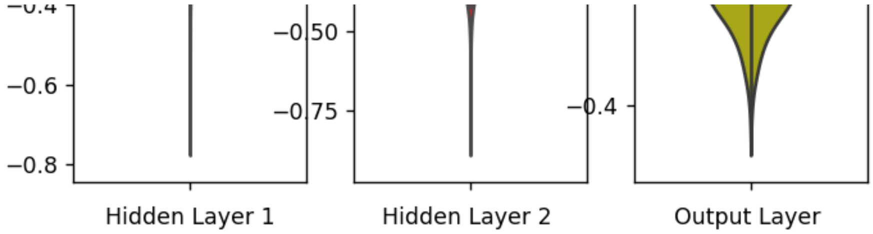
```
Test score: 0.09443054281140775
Test accuracy: 0.9819999933242798
```



In [18]:

```
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()
```





ARCHITECTURE 1= 784--612--324--10(2- Hidden layers)

B) MLP+Dropout+ADAM

In [19]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(612, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(324, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_5 (Dense)              (None, 612)               480420    
_________________________________________________________________
batch_normalization_1 (Batch (None, 612)               2448      
_________________________________________________________________
dropout_1 (Dropout)          (None, 612)               0         
_________________________________________________________________
dense_6 (Dense)              (None, 324)               198612    
_________________________________________________________________
batch_normalization_2 (Batch (None, 324)               1296      
_________________________________________________________________
dropout_2 (Dropout)          (None, 324)               0         
_________________________________________________________________
dense_7 (Dense)              (None, 10)                3250      
=================================================================
Total params: 686,026
Trainable params: 684,154
Non-trainable params: 1,872
_________________________________________________________________
```

In [20]:

```
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=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
```

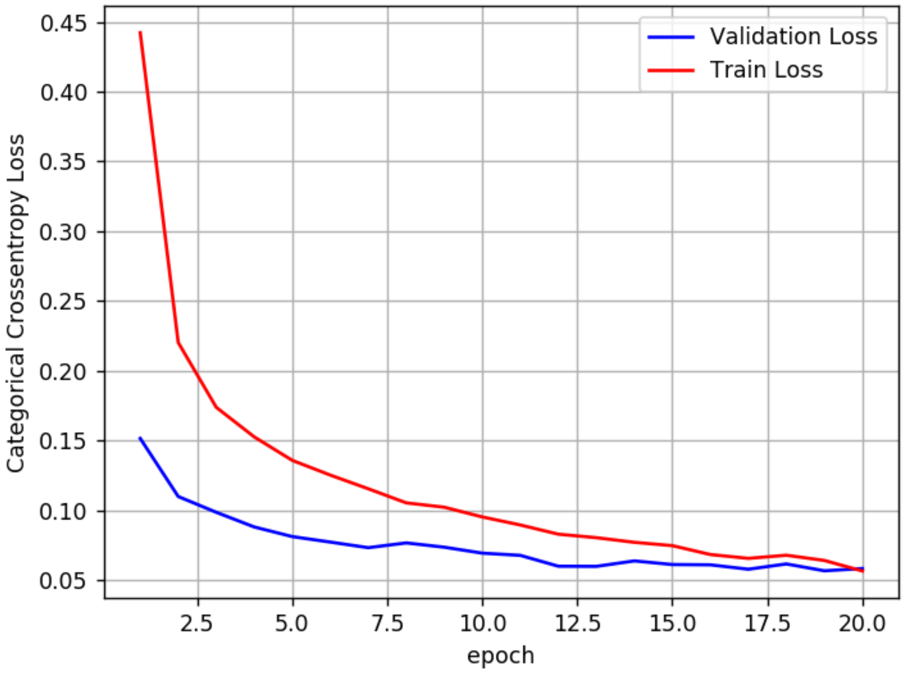
```
60000/60000 [==============================] - 3s 44us/step - loss: 0.4422 - accuracy: 0.8657 - va
l_loss: 0.1514 - val_accuracy: 0.9526
Epoch 2/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.2202 - accuracy: 0.9336 - va
l_loss: 0.1098 - val_accuracy: 0.9645
Epoch 3/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1737 - accuracy: 0.9467 - va
l_loss: 0.0985 - val_accuracy: 0.9685
Epoch 4/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.1525 - accuracy: 0.9529 - va
l_loss: 0.0879 - val_accuracy: 0.9716
Epoch 5/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1357 - accuracy: 0.9577 - va
l_loss: 0.0810 - val_accuracy: 0.9743
Epoch 6/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1251 - accuracy: 0.9608 - va
l_loss: 0.0771 - val_accuracy: 0.9766
Epoch 7/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.1153 - accuracy: 0.9640 - va
l_loss: 0.0731 - val_accuracy: 0.9770
Epoch 8/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1052 - accuracy: 0.9657 - va
l_loss: 0.0765 - val_accuracy: 0.9765
Epoch 9/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.1021 - accuracy: 0.9679 - va
l_loss: 0.0734 - val_accuracy: 0.9783
Epoch 10/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0951 - accuracy: 0.9706 - va
l_loss: 0.0692 - val_accuracy: 0.9790
Epoch 11/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0894 - accuracy: 0.9724 - va
l_loss: 0.0676 - val_accuracy: 0.9784
Epoch 12/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0828 - accuracy: 0.9741 - va
l_loss: 0.0598 - val_accuracy: 0.9821
Epoch 13/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0802 - accuracy: 0.9742 - va
l_loss: 0.0597 - val_accuracy: 0.9815
Epoch 14/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0769 - accuracy: 0.9757 - va
l_loss: 0.0636 - val_accuracy: 0.9802
Epoch 15/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0745 - accuracy: 0.9755 - va
l_loss: 0.0610 - val_accuracy: 0.9811
Epoch 16/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0682 - accuracy: 0.9787 - va
l_loss: 0.0608 - val_accuracy: 0.9802
Epoch 17/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0654 - accuracy: 0.9794 - va
l_loss: 0.0576 - val_accuracy: 0.9824
Epoch 18/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.0677 - accuracy: 0.9783 - va
l_loss: 0.0614 - val_accuracy: 0.9825
Epoch 19/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.0639 - accuracy: 0.9798 - va
l_loss: 0.0565 - val_accuracy: 0.9813
Epoch 20/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.0564 - accuracy: 0.9817 - va
l_loss: 0.0581 - val_accuracy: 0.9839
```

In [21]:

```python
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, verbose=1, va
lidation_data=(X_test, Y_test))
```

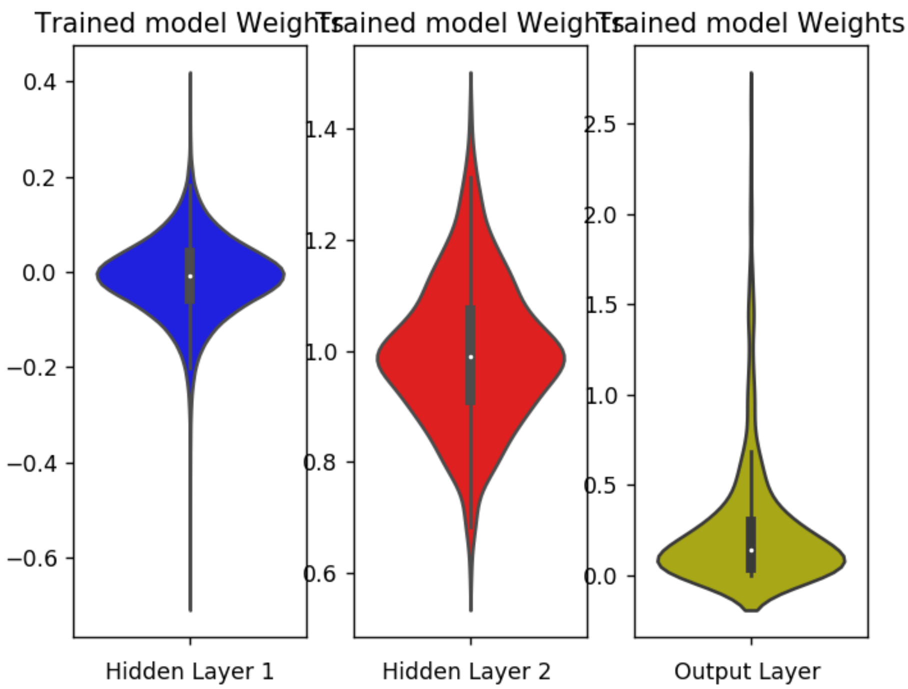
```
# we will get val_loss and val_acc only when you pass the paramter validation_data
# val_loss : validation loss
# val_acc : validation accuracy

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

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

```
Test score: 0.05811911709617998
Test accuracy: 0.9839000105857849
```



In [22]:

```
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()
```



ARCHITECTURE 2= 784--438--276--157--10(3- Hidden layers)

A) MLP+ReLu+ADAM

In [23]:

```
model_relu = Sequential()
model_relu.add(Dense(438, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(276, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125
, seed=None)) )
model_relu.add(Dense(157, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125
, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, valid
ation_data=(X_test, Y_test))
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 438)               343830
_________________________________________________________________
dense_9 (Dense)              (None, 276)               121164
_________________________________________________________________
dense_10 (Dense)             (None, 157)               43489
_________________________________________________________________
dense_11 (Dense)             (None, 10)                1580
=================================================================
Total params: 510,063
Trainable params: 510,063
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.2358 - accuracy: 0.9292 - va
l_loss: 0.1033 - val_accuracy: 0.9669
Epoch 2/20
```

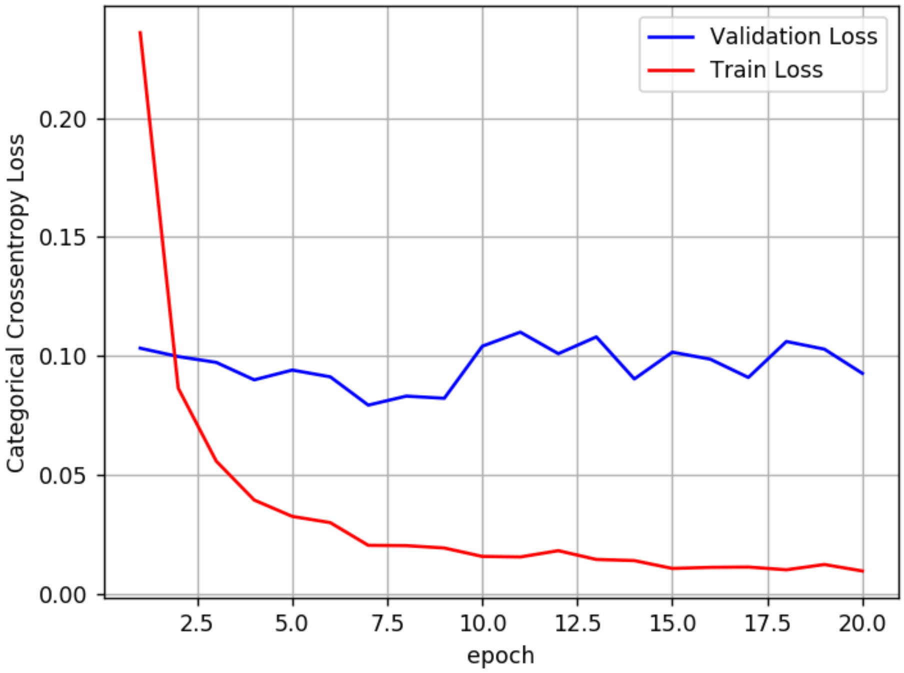
```
60000/60000 [==============================] - 2s 28us/step - loss: 0.0864 - accuracy: 0.9733 - va
l_loss: 0.0997 - val_accuracy: 0.9700
Epoch 3/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.0558 - accuracy: 0.9829 - va
l_loss: 0.0973 - val_accuracy: 0.9686
Epoch 4/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0394 - accuracy: 0.9875 - va
l_loss: 0.0900 - val_accuracy: 0.9736
Epoch 5/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0326 - accuracy: 0.9893 - va
l_loss: 0.0941 - val_accuracy: 0.9743
Epoch 6/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0300 - accuracy: 0.9898 - va
l_loss: 0.0912 - val_accuracy: 0.9716
Epoch 7/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0204 - accuracy: 0.9934 - va
l_loss: 0.0793 - val_accuracy: 0.9794
Epoch 8/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0203 - accuracy: 0.9933 - va
l_loss: 0.0831 - val_accuracy: 0.9792
Epoch 9/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0193 - accuracy: 0.9935 - va
l_loss: 0.0822 - val_accuracy: 0.9794
Epoch 10/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0158 - accuracy: 0.9943 - va
l_loss: 0.1041 - val_accuracy: 0.9764
Epoch 11/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0155 - accuracy: 0.9952 - va
l_loss: 0.1100 - val_accuracy: 0.9760
Epoch 12/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0182 - accuracy: 0.9941 - va
l_loss: 0.1010 - val_accuracy: 0.9794
Epoch 13/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.0145 - accuracy: 0.9949 - va
l_loss: 0.1080 - val_accuracy: 0.9768
Epoch 14/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.0140 - accuracy: 0.9951 - va
l_loss: 0.0903 - val_accuracy: 0.9819
Epoch 15/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.0107 - accuracy: 0.9967 - va
l_loss: 0.1016 - val_accuracy: 0.9823
Epoch 16/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.0112 - accuracy: 0.9964 - va
l_loss: 0.0986 - val_accuracy: 0.9800
Epoch 17/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.0113 - accuracy: 0.9963 - va
l_loss: 0.0910 - val_accuracy: 0.9808
Epoch 18/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.0101 - accuracy: 0.9965 - va
l_loss: 0.1061 - val_accuracy: 0.9797
Epoch 19/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.0124 - accuracy: 0.9964 - va
l_loss: 0.1029 - val_accuracy: 0.9783
Epoch 20/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.0096 - accuracy: 0.9970 - va
l_loss: 0.0927 - val_accuracy: 0.9814
```

In [24]:

```
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, verbose=1, va
lidation_data=(X_test, Y_test))

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

```
# val_acc : validation accuracy

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


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

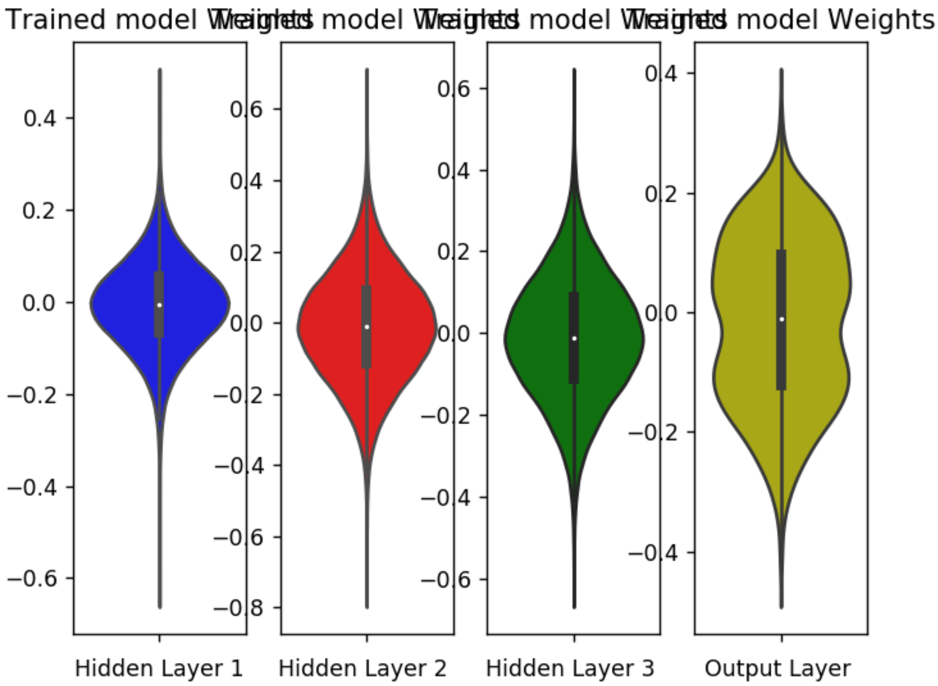
```
Test score: 0.09267379564880103
Test accuracy: 0.9814000129699707
```



In [25]:

```
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='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()
```



ARCHITECTURE 2= 784--438--276--157--10(3- Hidden layers)

B) MLP+Dropout+ADAM

In [26]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(438, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(276, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55,
seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(157, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_12 (Dense)             (None, 438)               343830    
_________________________________________________________________
batch_normalization_3 (Batch (None, 438)               1752      
```

```
_________________________________________________________________
dropout_3 (Dropout)          (None, 438)               0
_________________________________________________________________
dense_13 (Dense)             (None, 276)               121164
_________________________________________________________________
batch_normalization_4 (Batch (None, 276)               1104
_________________________________________________________________
dropout_4 (Dropout)          (None, 276)               0
_________________________________________________________________
dense_14 (Dense)             (None, 157)               43489
_________________________________________________________________
batch_normalization_5 (Batch (None, 157)               628
_________________________________________________________________
dropout_5 (Dropout)          (None, 157)               0
_________________________________________________________________
dense_15 (Dense)             (None, 10)                1580
=================================================================
Total params: 513,547
Trainable params: 511,805
Non-trainable params: 1,742
_________________________________________________________________
```

In [27]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, valid
ation_data=(X_test, Y_test))
```

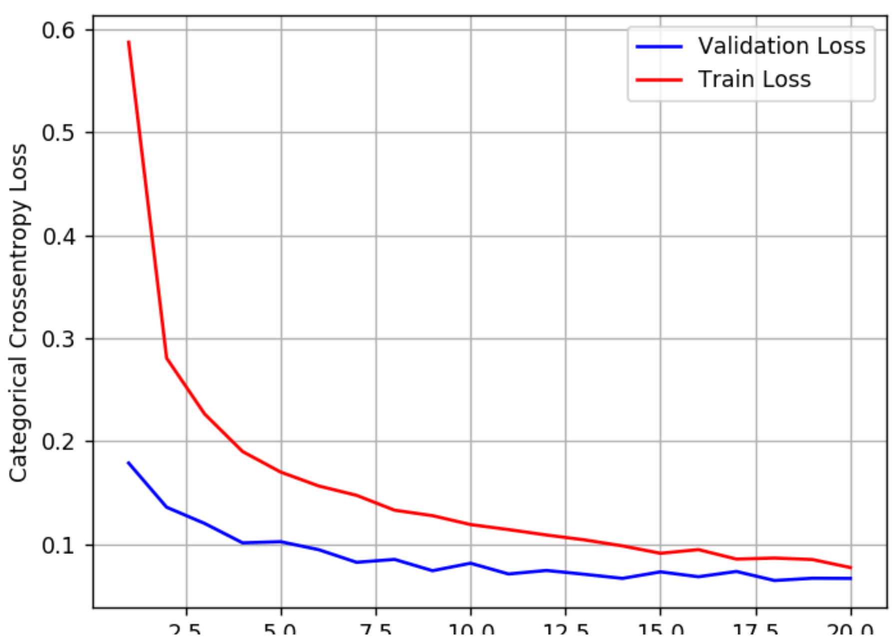
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.5875 - accuracy: 0.8211 - va
l_loss: 0.1787 - val_accuracy: 0.9440
Epoch 2/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.2806 - accuracy: 0.9160 - va
l_loss: 0.1358 - val_accuracy: 0.9572
Epoch 3/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.2264 - accuracy: 0.9320 - va
l_loss: 0.1200 - val_accuracy: 0.9633
Epoch 4/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1899 - accuracy: 0.9431 - va
l_loss: 0.1012 - val_accuracy: 0.9704
Epoch 5/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1700 - accuracy: 0.9491 - va
l_loss: 0.1023 - val_accuracy: 0.9704
Epoch 6/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1565 - accuracy: 0.9526 - va
l_loss: 0.0946 - val_accuracy: 0.9723
Epoch 7/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.1473 - accuracy: 0.9553 - va
l_loss: 0.0823 - val_accuracy: 0.9742
Epoch 8/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1329 - accuracy: 0.9597 - va
l_loss: 0.0851 - val_accuracy: 0.9756
Epoch 9/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1276 - accuracy: 0.9611 - va
l_loss: 0.0741 - val_accuracy: 0.9780
Epoch 10/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1190 - accuracy: 0.9638 - va
l_loss: 0.0814 - val_accuracy: 0.9759
Epoch 11/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.1141 - accuracy: 0.9650 - va
l_loss: 0.0710 - val_accuracy: 0.9794
Epoch 12/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1088 - accuracy: 0.9675 - va
l_loss: 0.0744 - val_accuracy: 0.9785
Epoch 13/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1041 - accuracy: 0.9686 - va
l_loss: 0.0707 - val_accuracy: 0.9790
Epoch 14/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0982 - accuracy: 0.9712 - va
l_loss: 0.0667 - val_accuracy: 0.9800
Epoch 15/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0910 - accuracy: 0.9724 - va
l_loss: 0.0730 - val_accuracy: 0.9782
```

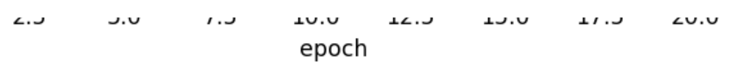
```
Epoch 16/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0946 - accuracy: 0.9720 - va
l_loss: 0.0683 - val_accuracy: 0.9796
Epoch 17/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0854 - accuracy: 0.9738 - va
l_loss: 0.0734 - val_accuracy: 0.9792
Epoch 18/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.0864 - accuracy: 0.9740 - va
l_loss: 0.0646 - val_accuracy: 0.9817
Epoch 19/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0850 - accuracy: 0.9746 - va
l_loss: 0.0668 - val_accuracy: 0.9816
Epoch 20/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0772 - accuracy: 0.9761 - va
l_loss: 0.0666 - val_accuracy: 0.9795
```

In [28]:

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

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

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

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, va
lidation_data=(X_test, Y_test))

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

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

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

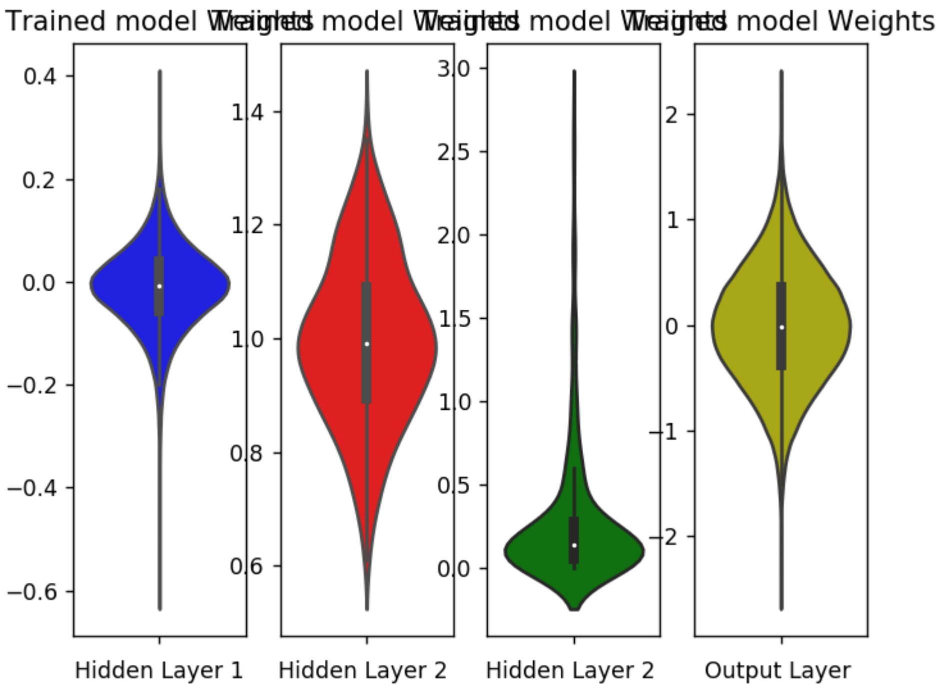
```
Test score: 0.0666387884610449
Test accuracy: 0.9794999957084656
```





In [29]:

```
w_after = model_drop.get_weights()

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

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



ARCHITECTURE 3= 784--551--447--331--236--121--10(5- Hidden layers)

A) MLP+ReLu+ADAM

In [30]:

```
model_relu = Sequential()
model_relu.add(Dense(551, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(447, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125
, seed=None)) )
model_relu.add(Dense(331, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125
, seed=None)) )
model_relu.add(Dense(236, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125
, seed=None)) )
model_relu.add(Dense(121, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125
, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, valid
ation_data=(X_test, Y_test))
```

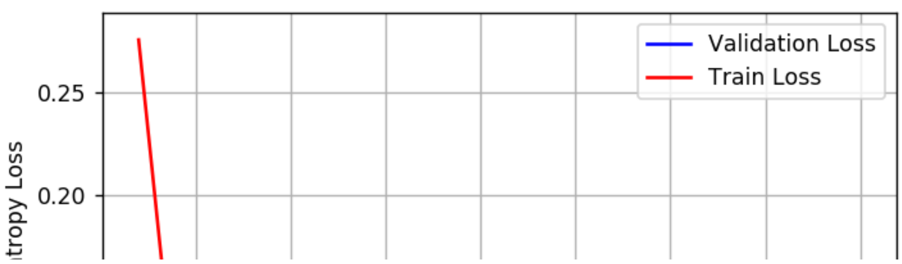
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_16 (Dense)             (None, 551)               432535    
_________________________________________________________________
dense_17 (Dense)             (None, 447)               246744    
_________________________________________________________________
dense_18 (Dense)             (None, 331)               148288    
_________________________________________________________________
dense_19 (Dense)             (None, 236)               78352     
_________________________________________________________________
dense_20 (Dense)             (None, 121)               28677     
_________________________________________________________________
dense_21 (Dense)             (None, 10)                1220      
=================================================================
Total params: 935,816
Trainable params: 935,816
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.2758 - accuracy: 0.9219 - va
l_loss: 0.1346 - val_accuracy: 0.9587
Epoch 2/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0971 - accuracy: 0.9699 - va
l_loss: 0.1024 - val_accuracy: 0.9675
Epoch 3/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0677 - accuracy: 0.9788 - va
l_loss: 0.1066 - val_accuracy: 0.9661
Epoch 4/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0525 - accuracy: 0.9829 - va
l_loss: 0.1103 - val_accuracy: 0.9695
Epoch 5/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0422 - accuracy: 0.9867 - va
l_loss: 0.0919 - val_accuracy: 0.9744
Epoch 6/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0366 - accuracy: 0.9878 - va
l_loss: 0.1137 - val_accuracy: 0.9713
Epoch 7/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0381 - accuracy: 0.9879 - va
l_loss: 0.1059 - val_accuracy: 0.9716
Epoch 8/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0298 - accuracy: 0.9902 - va
l_loss: 0.1128 - val_accuracy: 0.9710
Epoch 9/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0297 - accuracy: 0.9906 - va
l_loss: 0.1026 - val_accuracy: 0.9745
Epoch 10/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0269 - accuracy: 0.9912 - va
l_loss: 0.1038 - val_accuracy: 0.9738
Epoch 11/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0241 - accuracy: 0.9927 - va
```

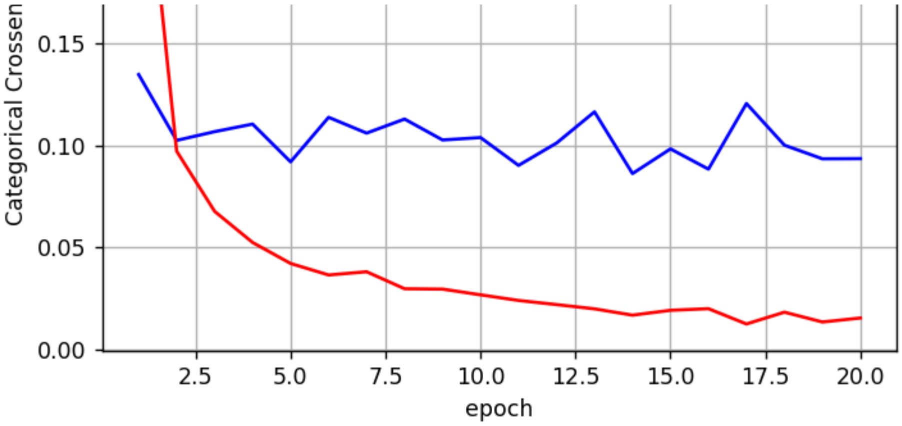
```
l_loss: 0.0901 - val_accuracy: 0.9800
Epoch 12/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0221 - accuracy: 0.9930 - va
l_loss: 0.1011 - val_accuracy: 0.9771
Epoch 13/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0200 - accuracy: 0.9937 - va
l_loss: 0.1163 - val_accuracy: 0.9735
Epoch 14/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0169 - accuracy: 0.9946 - va
l_loss: 0.0861 - val_accuracy: 0.9819
Epoch 15/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0193 - accuracy: 0.9947 - va
l_loss: 0.0982 - val_accuracy: 0.9776
Epoch 16/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0201 - accuracy: 0.9941 - va
l_loss: 0.0883 - val_accuracy: 0.9793
Epoch 17/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0126 - accuracy: 0.9964 - va
l_loss: 0.1204 - val_accuracy: 0.9753
Epoch 18/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0184 - accuracy: 0.9946 - va
l_loss: 0.1000 - val_accuracy: 0.9772
Epoch 19/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0136 - accuracy: 0.9960 - va
l_loss: 0.0934 - val_accuracy: 0.9814
Epoch 20/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0155 - accuracy: 0.9956 - va
l_loss: 0.0935 - val_accuracy: 0.9804
```

In [31]:

```
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, verbose=1, va
lidation_data=(X_test, Y_test))

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

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


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

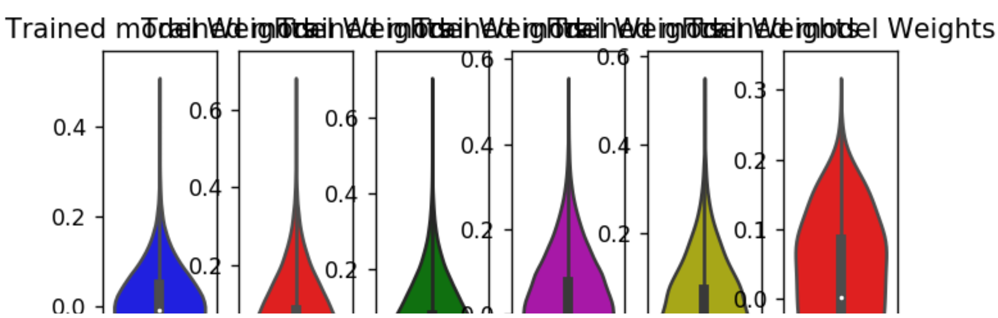
```
Test score: 0.09345374069254558
Test accuracy: 0.980400025844574
```

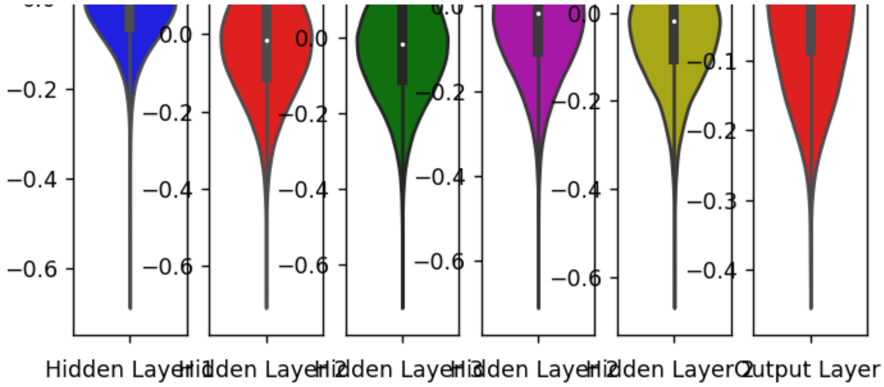




In [32]:

```
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\n")
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='m')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h5_w, color='y')
plt.xlabel('Hidden Layer 2 ')

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





ARCHITECTURE 3= 784--551--447--331--236--121--10(5- Hidden layers)

B) MLP+Dropout+ADAM

In [33]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(551, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(447, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(331, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(236, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(121, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_22 (Dense)             (None, 551)               432535
_________________________________________________________________
batch_normalization_6 (Batch (None, 551)               2204
_________________________________________________________________
dropout_6 (Dropout)          (None, 551)               0
_________________________________________________________________
dense_23 (Dense)             (None, 447)               246744
```

```
_________________________________________________________________
batch_normalization_7 (Batch (None, 447)               1788
_________________________________________________________________
dropout_7 (Dropout)          (None, 447)               0
_________________________________________________________________
dense_24 (Dense)             (None, 331)               148288
_________________________________________________________________
batch_normalization_8 (Batch (None, 331)               1324
_________________________________________________________________
dropout_8 (Dropout)          (None, 331)               0
_________________________________________________________________
dense_25 (Dense)             (None, 236)               78352
_________________________________________________________________
batch_normalization_9 (Batch (None, 236)               944
_________________________________________________________________
dropout_9 (Dropout)          (None, 236)               0
_________________________________________________________________
dense_26 (Dense)             (None, 121)               28677
_________________________________________________________________
batch_normalization_10 (Batc (None, 121)               484
_________________________________________________________________
dropout_10 (Dropout)         (None, 121)               0
_________________________________________________________________
dense_27 (Dense)             (None, 10)                1220
=================================================================
Total params: 942,560
Trainable params: 939,188
Non-trainable params: 3,372
_________________________________________________________________
```

In [34]:

```
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=1, valid
ation_data=(X_test, Y_test))
```

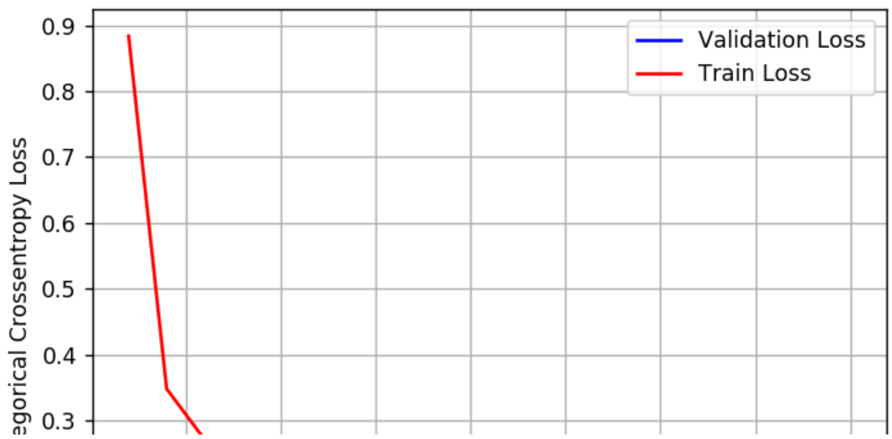
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.8836 - accuracy: 0.7288 - va
l_loss: 0.2308 - val_accuracy: 0.9279
Epoch 2/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.3485 - accuracy: 0.8990 - va
l_loss: 0.1723 - val_accuracy: 0.9492
Epoch 3/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.2714 - accuracy: 0.9225 - va
l_loss: 0.1403 - val_accuracy: 0.9592
Epoch 4/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.2290 - accuracy: 0.9353 - va
l_loss: 0.1156 - val_accuracy: 0.9666
Epoch 5/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.2026 - accuracy: 0.9427 - va
l_loss: 0.1083 - val_accuracy: 0.9691
Epoch 6/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1872 - accuracy: 0.9463 - va
l_loss: 0.1063 - val_accuracy: 0.9708
Epoch 7/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1725 - accuracy: 0.9505 - va
l_loss: 0.1015 - val_accuracy: 0.9709
Epoch 8/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1624 - accuracy: 0.9543 - va
l_loss: 0.0982 - val_accuracy: 0.9715
Epoch 9/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1524 - accuracy: 0.9571 - va
l_loss: 0.0867 - val_accuracy: 0.9752
Epoch 10/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1413 - accuracy: 0.9604 - va
l_loss: 0.0796 - val_accuracy: 0.9793
Epoch 11/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1354 - accuracy: 0.9614 - va
l_loss: 0.0797 - val_accuracy: 0.9774
Epoch 12/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1296 - accuracy: 0.9629 - va
l_loss: 0.0803 - val_accuracy: 0.9782
Epoch 13/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1253 - accuracy: 0.9645 - va
```

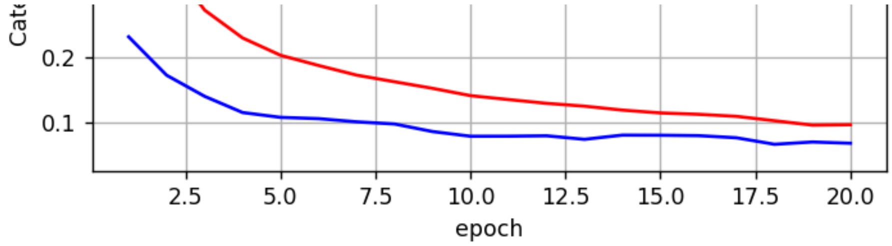
```
l_loss: 0.0749 - val_accuracy: 0.9798
Epoch 14/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1193 - accuracy: 0.9658 - va
l_loss: 0.0813 - val_accuracy: 0.9787
Epoch 15/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1150 - accuracy: 0.9681 - va
l_loss: 0.0812 - val_accuracy: 0.9786
Epoch 16/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1129 - accuracy: 0.9677 - va
l_loss: 0.0805 - val_accuracy: 0.9786
Epoch 17/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1098 - accuracy: 0.9683 - va
l_loss: 0.0773 - val_accuracy: 0.9792
Epoch 18/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1032 - accuracy: 0.9712 - va
l_loss: 0.0674 - val_accuracy: 0.9822
Epoch 19/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0965 - accuracy: 0.9728 - va
l_loss: 0.0708 - val_accuracy: 0.9811
Epoch 20/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0971 - accuracy: 0.9725 - va
l_loss: 0.0690 - val_accuracy: 0.9806
```

In [35]:

```python
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, verbose=1, va
lidation_data=(X_test, Y_test))

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

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

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

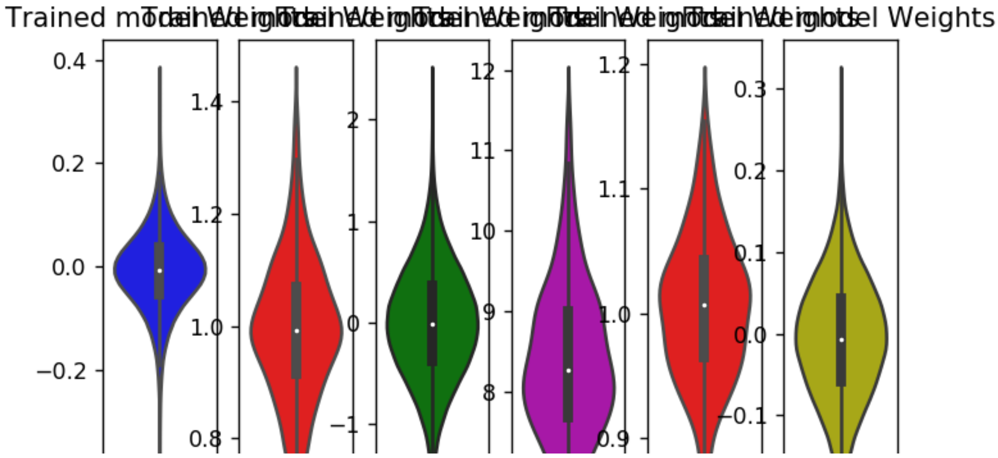
```
Test score: 0.06895258962376974
Test accuracy: 0.9805999994277954
```

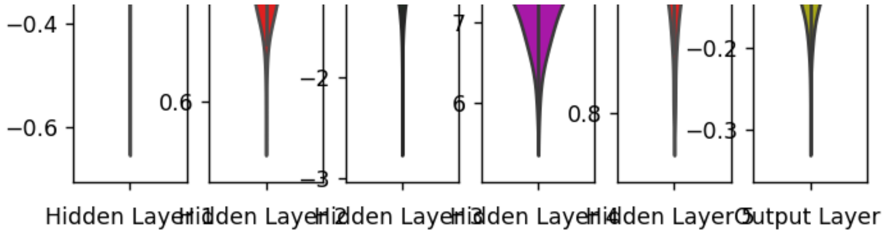




In [36]:

```
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[6].flatten().reshape(-1,1)
h4_w = w_after[10].flatten().reshape(-1,1)
h5_w = w_after[14].flatten().reshape(-1,1)
out_w = w_after[18].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='m')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h5_w, color='r')
plt.xlabel('Hidden Layer 5 ')

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




−0.4
−0.6
0.6
−2
−3
7
6
0.8
−0.2
−0.3
Hidden Layer 1
Hidden Layer 2
Hidden Layer 3
Hidden Layer 4
Hidden Layer 5
Output Layer

In [ ]: