Keras -- MLPs on MNIST

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

#install the nightly artifacts of tensorflow in order to use TensorFlow eager mod
e. It is a new, experimental feature that is not yet included in the releases.
```

```
Using TensorFlow backend.
C:\Users\wwang26\AppData\Local\Continuum\anaconda3\lib\site-packages\tensorflow
\python\framework\dtypes.py:526: FutureWarning: Passing (type, 1) or '1type' as
a synonym of type is deprecated; in a future version of numpy, it will be unders
tood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
C:\Users\wwang26\AppData\Local\Continuum\anaconda3\lib\site-packages\tensorflow
\python\framework\dtypes.py:527: FutureWarning: Passing (type, 1) or '1type' as
a synonym of type is deprecated; in a future version of numpy, it will be unders
tood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
C:\Users\wwang26\AppData\Local\Continuum\anaconda3\lib\site-packages\tensorflow
\python\framework\dtypes.py:528: FutureWarning: Passing (type, 1) or '1type' as
a synonym of type is deprecated; in a future version of numpy, it will be unders
tood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
C:\Users\wwang26\AppData\Local\Continuum\anaconda3\lib\site-packages\tensorflow
\python\framework\dtypes.py:529: FutureWarning: Passing (type, 1) or '1type' as
a synonym of type is deprecated; in a future version of numpy, it will be unders
tood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
C:\Users\wwang26\AppData\Local\Continuum\anaconda3\lib\site-packages\tensorflow
\python\framework\dtypes.py:530: FutureWarning: Passing (type, 1) or '1type' as
a synonym of type is deprecated; in a future version of numpy, it will be unders
tood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
C:\Users\wwang26\AppData\Local\Continuum\anaconda3\lib\site-packages\tensorflow
\python\framework\dtypes.py:535: FutureWarning: Passing (type, 1) or '1type' as
a synonym of type is deprecated; in a future version of numpy, it will be unders
tood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

In [2]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
import time
# https://gist.github.com/greydanus/f6eee59eaf1d90fcb3b534a25362cea4
# https://stackoverflow.com/a/14434334
# this function is used to update the plots for each epoch and error
def plt_dynamic(x, vy, ty, ax, colors=['b']):
    ax.plot(x, vy, 'b', label="Validation Loss")
    ax.plot(x, ty, 'r', label="Train Loss")
    plt.legend()
    plt.grid()
    fig.canvas.draw()
```

In [3]:
```
# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [==============================] - 1s 0us/step
```

In [4]:
```
print("Number of training examples :", X_train.shape[0], "and each image is of sh
ape (%d, %d)"%(X_train.shape[1], X_train.shape[2]))
print("Number of training examples :", X_test.shape[0], "and each image is of sha
pe (%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 sh
ape (%d)"%(X_train.shape[1]))
print("Number of training examples :", X_test.shape[0], "and each image is of sha
pe (%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 d
ata
# X => (X - Xmin)/(Xmax-Xmin) = X/255

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

In [9]:
```
# example data point after normlizing
print(X_train[0])
```

/

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

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

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

In [10]:

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

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

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

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

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

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 c
onstructor:

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

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

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

###

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

# keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='g
lorot_uniform',
# bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activ
ity_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 arg
ument,
# 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 Tru
e).

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

####

# https://keras.io/activations/

# Activations can either be used through an Activation layer, or through the acti
vation argument supported by all forward layers:

# from keras.layers import Activation, Dense

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

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

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


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

In [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'))
```

```
WARNING:tensorflow:From C:\Users\wwang26\AppData\Local\Continuum\anaconda3\lib\s
ite-packages\tensorflow\python\ops\resource_variable_ops.py:435: colocate_with
(from tensorflow.python.framework.ops) is deprecated and will be removed in a fu
ture version.
Instructions for updating:
Colocations handled automatically by placer.
```

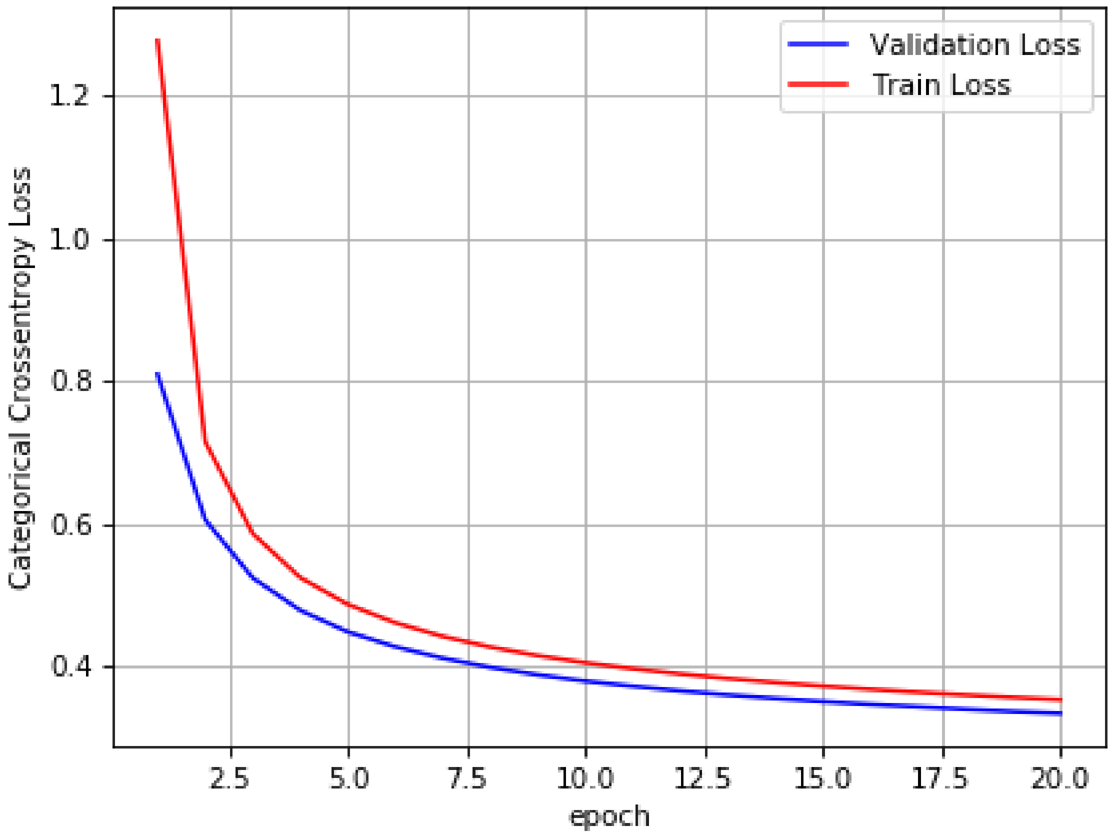
In [14]:
```
# Before training a model, you need to configure the learning process, which is d
one via the compile method

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


# Note: when using the categorical_crossentropy loss, your targets should be in c
ategorical format
# (e.g. if you have 10 classes, the target for each sample should be a 10-dimensi
onal 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='sgd', loss='categorical_crossentropy', metrics=['accurac
y'])

# 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, init
ial_epoch=0, steps_per_epoch=None,
# validation_steps=None)

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

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

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

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

```
WARNING:tensorflow:From C:\Users\wwang26\AppData\Local\Continuum\anaconda3\lib\s
ite-packages\tensorflow\python\ops\math_ops.py:3066: to_int32 (from tensorflow.p
ython.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 35us/step - loss: 1.2756 - acc
uracy: 0.6960 - val_loss: 0.8088 - val_accuracy: 0.8340
Epoch 2/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.7148 - acc
uracy: 0.8408 - val_loss: 0.6054 - val_accuracy: 0.8616
Epoch 3/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.5861 - acc
uracy: 0.8606 - val_loss: 0.5241 - val_accuracy: 0.8731
Epoch 4/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.5246 - acc
uracy: 0.8695 - val_loss: 0.4787 - val_accuracy: 0.8796
Epoch 5/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.4871 - acc
uracy: 0.8763 - val_loss: 0.4485 - val_accuracy: 0.8862
Epoch 6/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.4613 - acc
uracy: 0.8805 - val_loss: 0.4277 - val_accuracy: 0.8888
Epoch 7/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.4422 - acc
uracy: 0.8841 - val_loss: 0.4117 - val_accuracy: 0.8920
Epoch 8/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.4273 - acc
uracy: 0.8870 - val_loss: 0.3988 - val_accuracy: 0.8950
Epoch 9/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.4153 - acc
uracy: 0.8893 - val_loss: 0.3883 - val_accuracy: 0.8972
Epoch 10/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.4053 - acc
uracy: 0.8913 - val_loss: 0.3795 - val_accuracy: 0.8988
Epoch 11/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3969 - acc
uracy: 0.8929 - val_loss: 0.3723 - val_accuracy: 0.9005
Epoch 12/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.3896 - acc
uracy: 0.8948 - val_loss: 0.3659 - val_accuracy: 0.9013
Epoch 13/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.3832 - acc
uracy: 0.8959 - val_loss: 0.3603 - val_accuracy: 0.9033
Epoch 14/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3776 - acc
uracy: 0.8974 - val_loss: 0.3554 - val_accuracy: 0.9047
Epoch 15/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.3726 - acc
uracy: 0.8982 - val_loss: 0.3511 - val_accuracy: 0.9055
Epoch 16/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.3681 - acc
uracy: 0.8991 - val_loss: 0.3471 - val_accuracy: 0.9064
Epoch 17/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.3640 - acc
uracy: 0.9005 - val_loss: 0.3438 - val_accuracy: 0.9070
Epoch 18/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.3603 - acc
```

```
uracy: 0.9012 - val_loss: 0.3404 - val_accuracy: 0.9073
Epoch 19/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.3569 - acc
uracy: 0.9019 - val_loss: 0.3372 - val_accuracy: 0.9077
Epoch 20/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.3537 - acc
uracy: 0.9026 - val_loss: 0.3346 - val_accuracy: 0.9094
```

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_epo
ch, verbose=1, validation_data=(X_test, Y_test))

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.3346195185959339
Test accuracy: 0.9093999862670898
```



MLP + Sigmoid activation + SGDOptimizer

In [16]:
```
# Multilayer perceptron

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

model_sigmoid.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_2 (Dense)              (None, 512)               401920    
_________________________________________________________________
dense_3 (Dense)              (None, 128)               65664     
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

In [17]:
```
model_sigmoid.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=[
'accuracy'])

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_ep
och, verbose=1, validation_data=(X_test, Y_test))
```

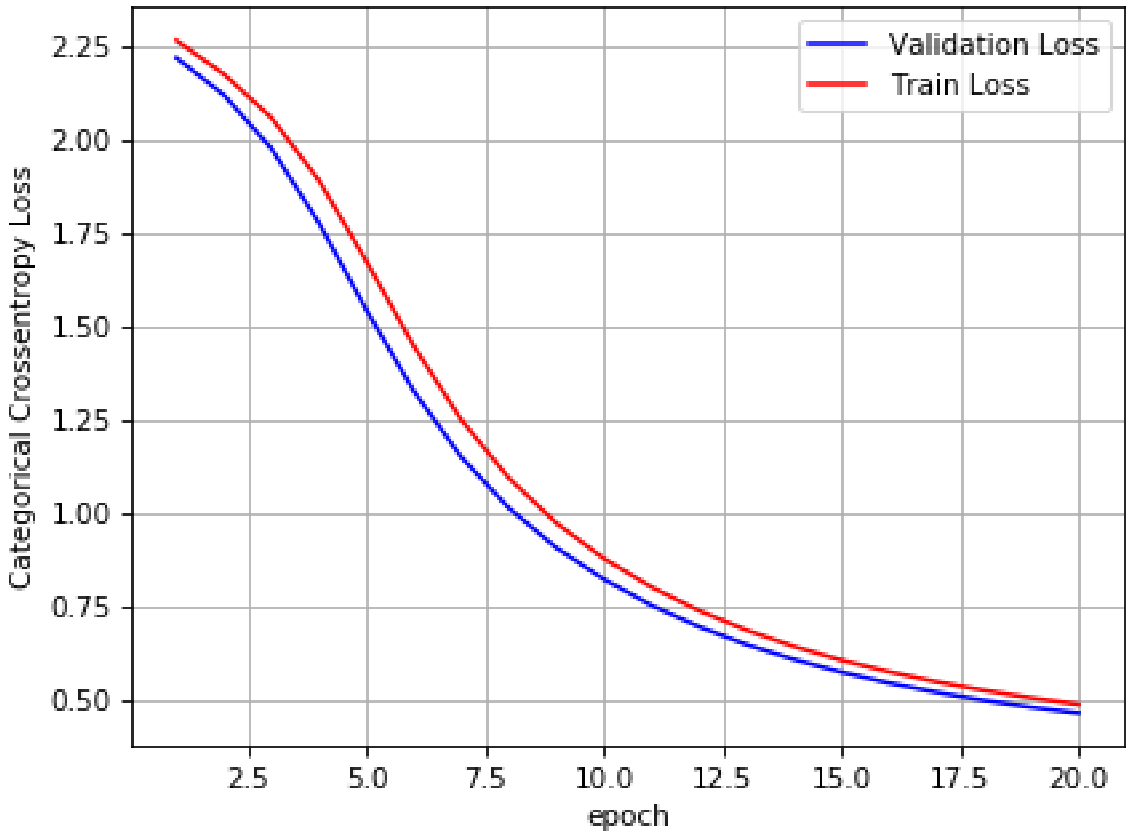
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 9s 151us/step - loss: 2.2688 - ac
curacy: 0.2144 - val_loss: 2.2232 - val_accuracy: 0.4541
Epoch 2/20
60000/60000 [==============================] - 10s 174us/step - loss: 2.1797 - a
ccuracy: 0.4497 - val_loss: 2.1238 - val_accuracy: 0.4339
Epoch 3/20
60000/60000 [==============================] - 9s 150us/step - loss: 2.0630 - ac
curacy: 0.5544 - val_loss: 1.9805 - val_accuracy: 0.6145
Epoch 4/20
60000/60000 [==============================] - 9s 149us/step - loss: 1.8953 - ac
curacy: 0.6119 - val_loss: 1.7814 - val_accuracy: 0.6170
Epoch 5/20
60000/60000 [==============================] - 8s 140us/step - loss: 1.6786 - ac
curacy: 0.6469 - val_loss: 1.5481 - val_accuracy: 0.6888
Epoch 6/20
60000/60000 [==============================] - 7s 118us/step - loss: 1.4512 - ac
curacy: 0.6865 - val_loss: 1.3303 - val_accuracy: 0.7214
Epoch 7/20
60000/60000 [==============================] - 8s 137us/step - loss: 1.2536 - ac
curacy: 0.7221 - val_loss: 1.1537 - val_accuracy: 0.7373
Epoch 8/20
60000/60000 [==============================] - 10s 170us/step - loss: 1.0980 - a
ccuracy: 0.7494 - val_loss: 1.0176 - val_accuracy: 0.7702
Epoch 9/20
60000/60000 [==============================] - 8s 140us/step - loss: 0.9772 - ac
curacy: 0.7732 - val_loss: 0.9108 - val_accuracy: 0.7908
Epoch 10/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.8821 - a
ccuracy: 0.7913 - val_loss: 0.8266 - val_accuracy: 0.8103
Epoch 11/20
60000/60000 [==============================] - 13s 214us/step - loss: 0.8054 - a
ccuracy: 0.8066 - val_loss: 0.7563 - val_accuracy: 0.8218
Epoch 12/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.7424 - a
ccuracy: 0.8199 - val_loss: 0.6992 - val_accuracy: 0.8293
Epoch 13/20
60000/60000 [==============================] - 15s 248us/step - loss: 0.6901 - a
ccuracy: 0.8300 - val_loss: 0.6514 - val_accuracy: 0.8430
Epoch 14/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.6463 - ac
curacy: 0.8381 - val_loss: 0.6114 - val_accuracy: 0.8497
Epoch 15/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.6095 - ac
curacy: 0.8462 - val_loss: 0.5774 - val_accuracy: 0.8550
Epoch 16/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.5785 - ac
curacy: 0.8523 - val_loss: 0.5484 - val_accuracy: 0.8613
Epoch 17/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.5517 - ac
curacy: 0.8578 - val_loss: 0.5238 - val_accuracy: 0.8669
Epoch 18/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.5289 - ac
curacy: 0.8636 - val_loss: 0.5025 - val_accuracy: 0.8689
Epoch 19/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.5091 - ac
curacy: 0.8673 - val_loss: 0.4841 - val_accuracy: 0.8736
Epoch 20/20
```

```
60000/60000 [==============================] - 8s 136us/step - loss: 0.4917 - ac
curacy: 0.8706 - val_loss: 0.4683 - val_accuracy: 0.8748
```

In [18]:

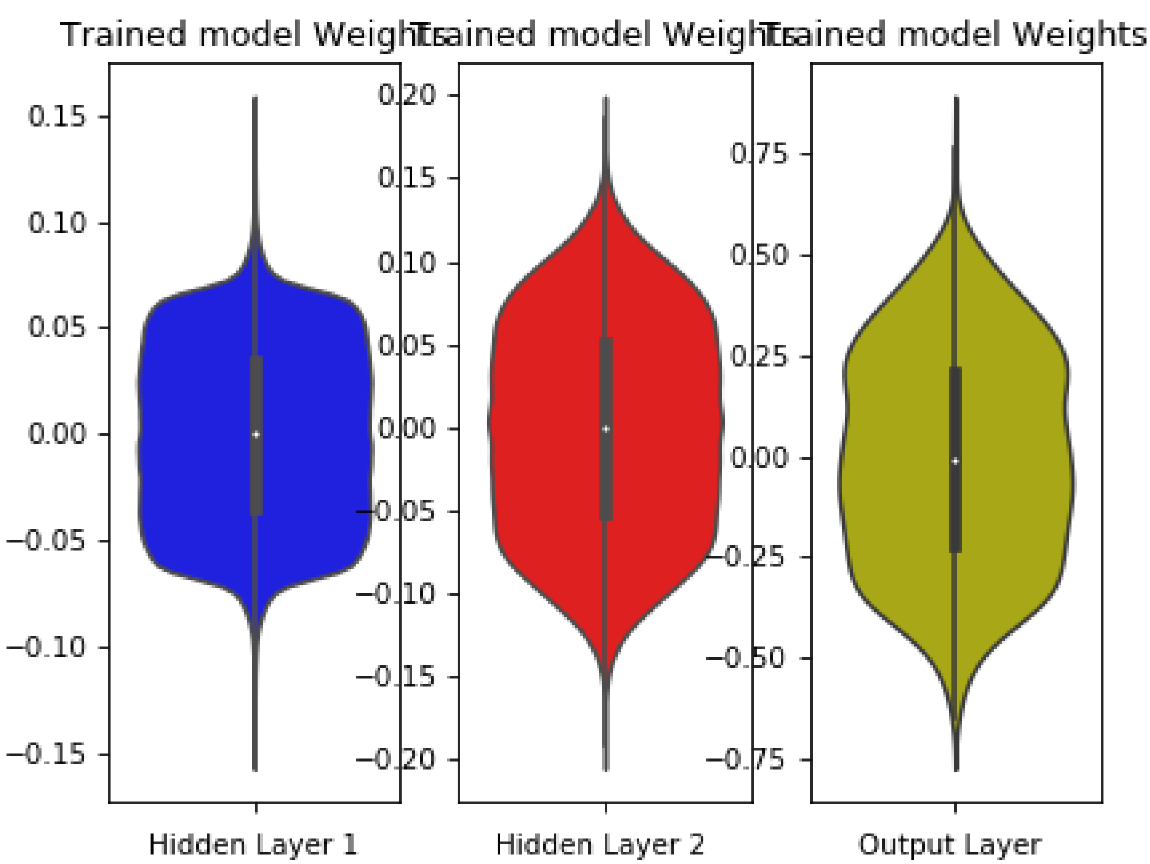
```
score = model_sigmoid.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.468309353351593
Test accuracy: 0.8748000264167786
```



/

In [20]:
```
w_after = model_sigmoid.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 + Sigmoid activation + ADAM

/

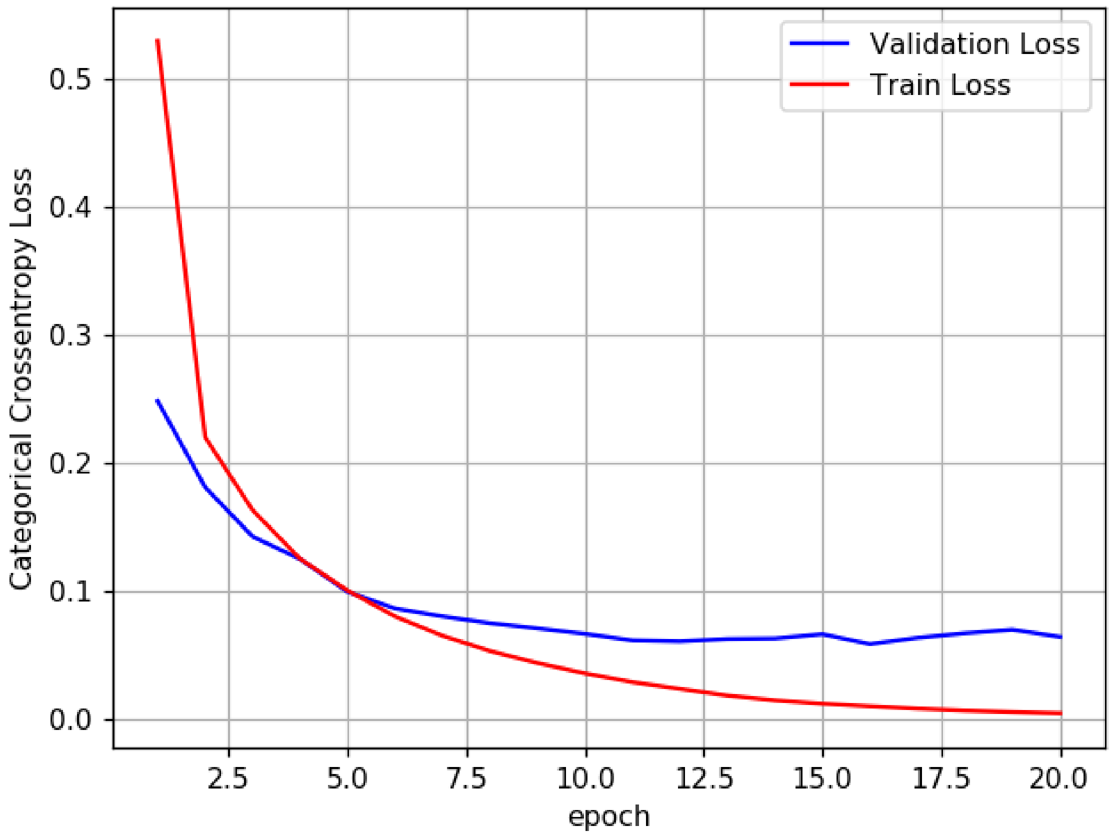
In [0]:
```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shape=(input_dim,)))
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()

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

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_ep
och, verbose=1, validation_data=(X_test, Y_test))
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_6 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.5308 - ac
c: 0.8636 - val_loss: 0.2550 - val_acc: 0.9259
Epoch 2/20
53120/60000 [=========================>....] - ETA: 0s - loss: 0.2244 - acc: 0.9
34060000/60000 [==============================] - 3s 51us/step - loss: 0.2205 -
acc: 0.9351 - val_loss: 0.1946 - val_acc: 0.9417
Epoch 3/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.1650 - ac
c: 0.9512 - val_loss: 0.1421 - val_acc: 0.9570
Epoch 4/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.1279 - ac
c: 0.9614 - val_loss: 0.1238 - val_acc: 0.9645
Epoch 5/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.1010 - ac
c: 0.9704 - val_loss: 0.1029 - val_acc: 0.9693
Epoch 6/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0801 - ac
c: 0.9763 - val_loss: 0.0877 - val_acc: 0.9725
Epoch 7/20
 4480/60000 [=>............................] - ETA: 2s - loss: 0.0632 - acc: 0.9
79560000/60000 [==============================] - 3s 51us/step - loss: 0.0645 -
acc: 0.9809 - val_loss: 0.0831 - val_acc: 0.9751
Epoch 8/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0519 - ac
c: 0.9842 - val_loss: 0.0724 - val_acc: 0.9780
Epoch 9/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0433 - ac
c: 0.9872 - val_loss: 0.0714 - val_acc: 0.9786
Epoch 10/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0347 - ac
c: 0.9898 - val_loss: 0.0695 - val_acc: 0.9776
Epoch 11/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0268 - ac
c: 0.9930 - val_loss: 0.0659 - val_acc: 0.9796
Epoch 12/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.0219 - ac
c: 0.9944 - val_loss: 0.0642 - val_acc: 0.9809
Epoch 13/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0180 - ac
c: 0.9953 - val_loss: 0.0677 - val_acc: 0.9794
Epoch 14/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0133 - ac
c: 0.9970 - val_loss: 0.0647 - val_acc: 0.9803
Epoch 15/20
```

```
60000/60000 [==============================] - 3s 50us/step - loss: 0.0114 - ac
c: 0.9975 - val_loss: 0.0628 - val_acc: 0.9812
Epoch 16/20
57344/60000 [===========================>..] - ETA: 0s - loss: 0.0085 - acc: 0.9
98260000/60000 [==============================] - 3s 50us/step - loss: 0.0085 -
acc: 0.9982 - val_loss: 0.0666 - val_acc: 0.9806
Epoch 17/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0070 - ac
c: 0.9986 - val_loss: 0.0643 - val_acc: 0.9822
Epoch 18/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0061 - ac
c: 0.9986 - val_loss: 0.0656 - val_acc: 0.9818
Epoch 19/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.0055 - ac
c: 0.9988 - val_loss: 0.0811 - val_acc: 0.9774
Epoch 20/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0038 - ac
c: 0.9992 - val_loss: 0.0723 - val_acc: 0.9818
```

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

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.06385514608082886
Test accuracy: 0.9824
```



/

In [0]:
```
w_after = model_sigmoid.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()
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:588: FutureWarnin
g: remove_na is deprecated and is a private function. Do not use.
  kde_data = remove_na(group_data)
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:816: FutureWarnin
g: remove_na is deprecated and is a private function. Do not use.
  violin_data = remove_na(group_data)
```

MLP + ReLU +SGD

In [0]:
```
# Multilayer perceptron

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

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

model_relu.summary()
```

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

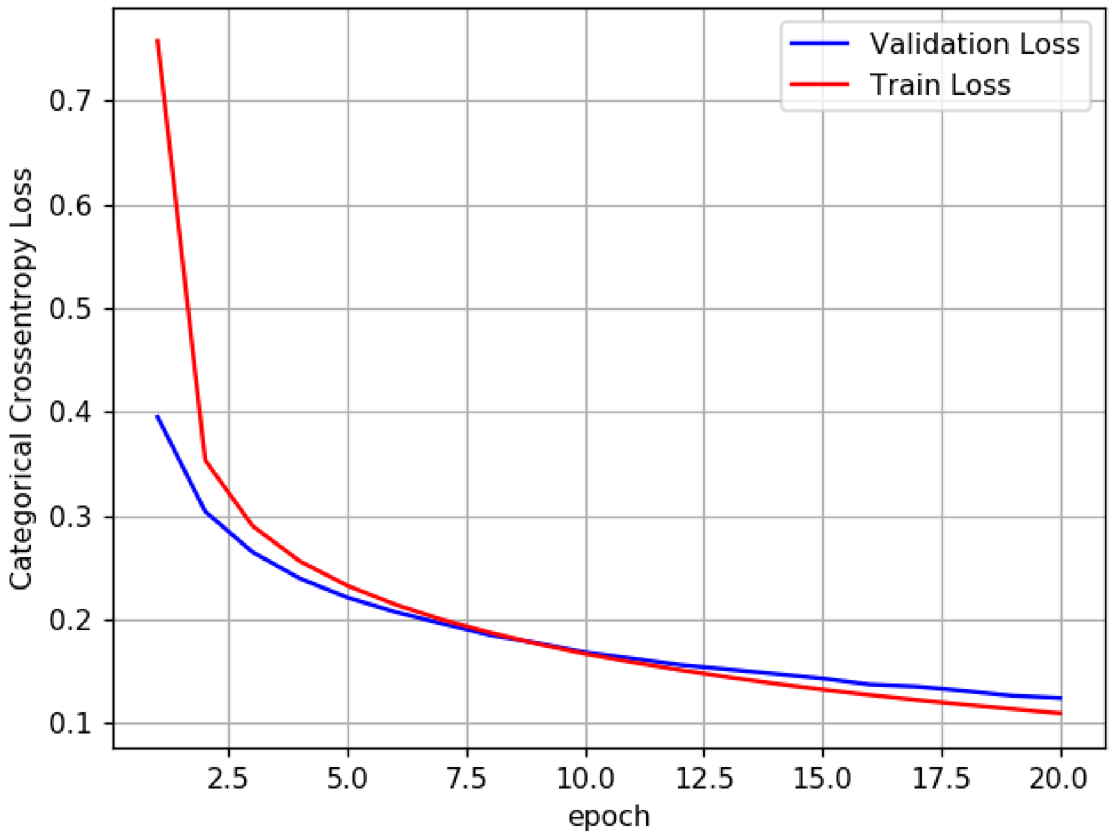
In [0]:
```
model_relu.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['ac
curacy'])

history = model_relu.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 [==============================] - 4s 67us/step - loss: 0.7579 - ac
c: 0.7812 - val_loss: 0.3951 - val_acc: 0.8921
Epoch 2/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.3535 - ac
c: 0.8998 - val_loss: 0.3040 - val_acc: 0.9153
Epoch 3/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.2900 - ac
c: 0.9172 - val_loss: 0.2648 - val_acc: 0.9253
Epoch 4/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.2558 - ac
c: 0.9269 - val_loss: 0.2393 - val_acc: 0.9316
Epoch 5/20
60000/60000 [==============================] - 4s 58us/step - loss: 0.2324 - ac
c: 0.9340 - val_loss: 0.2210 - val_acc: 0.9371
Epoch 6/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.2144 - ac
c: 0.9391 - val_loss: 0.2072 - val_acc: 0.9400
Epoch 7/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1995 - ac
c: 0.9443 - val_loss: 0.1957 - val_acc: 0.9444
Epoch 8/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.1872 - ac
c: 0.9476 - val_loss: 0.1848 - val_acc: 0.9456
Epoch 9/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.1763 - ac
c: 0.9507 - val_loss: 0.1771 - val_acc: 0.9488
Epoch 10/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.1668 - ac
c: 0.9539 - val_loss: 0.1682 - val_acc: 0.9506
Epoch 11/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.1585 - ac
c: 0.9560 - val_loss: 0.1623 - val_acc: 0.9518
Epoch 12/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.1511 - ac
c: 0.9577 - val_loss: 0.1560 - val_acc: 0.9543
Epoch 13/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.1443 - ac
c: 0.9596 - val_loss: 0.1517 - val_acc: 0.9557
Epoch 14/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.1379 - ac
c: 0.9615 - val_loss: 0.1474 - val_acc: 0.9572
Epoch 15/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1323 - ac
c: 0.9628 - val_loss: 0.1429 - val_acc: 0.9580
Epoch 16/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1270 - ac
c: 0.9645 - val_loss: 0.1371 - val_acc: 0.9598
Epoch 17/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.1221 - ac
c: 0.9661 - val_loss: 0.1351 - val_acc: 0.9602
Epoch 18/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.1177 - ac
c: 0.9671 - val_loss: 0.1309 - val_acc: 0.9618
Epoch 19/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.1136 - ac
c: 0.9685 - val_loss: 0.1263 - val_acc: 0.9631
Epoch 20/20
```

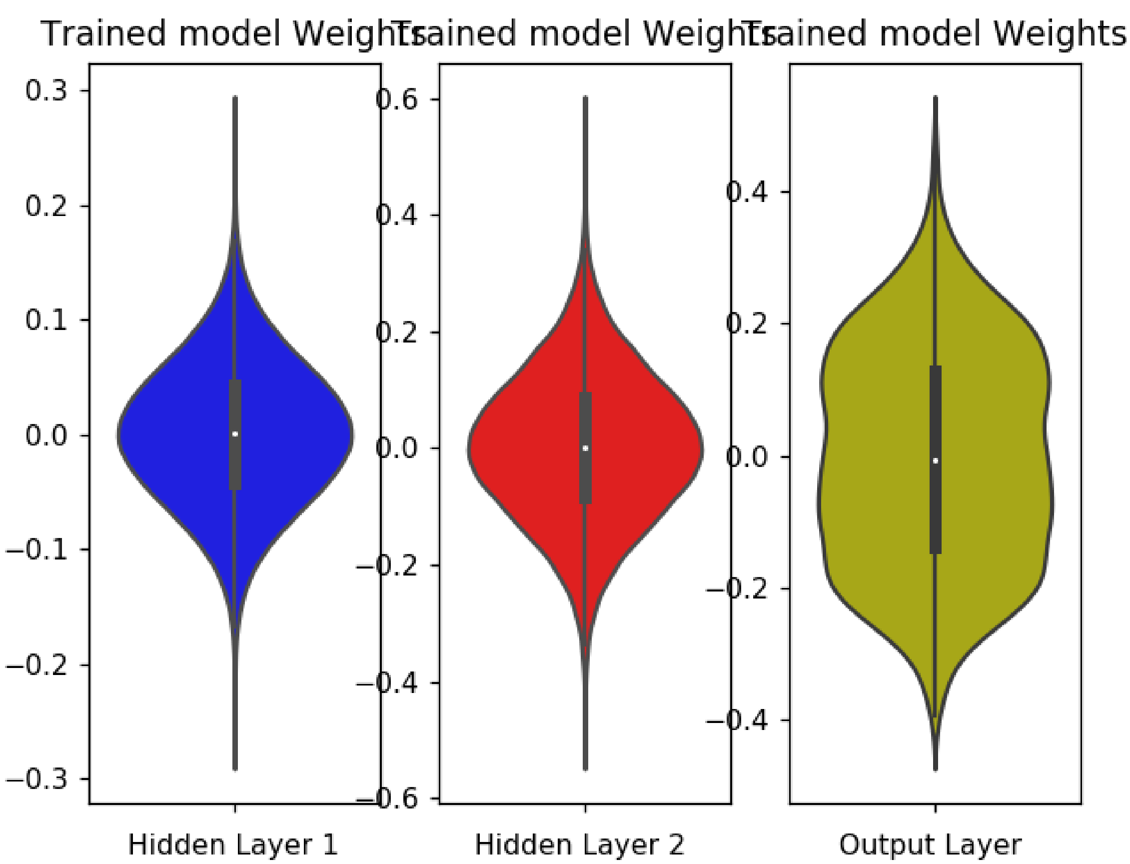
```
60000/60000 [==============================] - 5s 79us/step - loss: 0.1094 - ac
c: 0.9694 - val_loss: 0.1241 - val_acc: 0.9631
```

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

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.12405014228336513
Test accuracy: 0.9631
```



In [0]:
```
w_after = model_relu.get_weights()

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


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

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

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



MLP + ReLU + ADAM

/

In [0]:
```
model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_ini
tializer=RandomNormal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean
=0.0, stddev=0.125, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

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

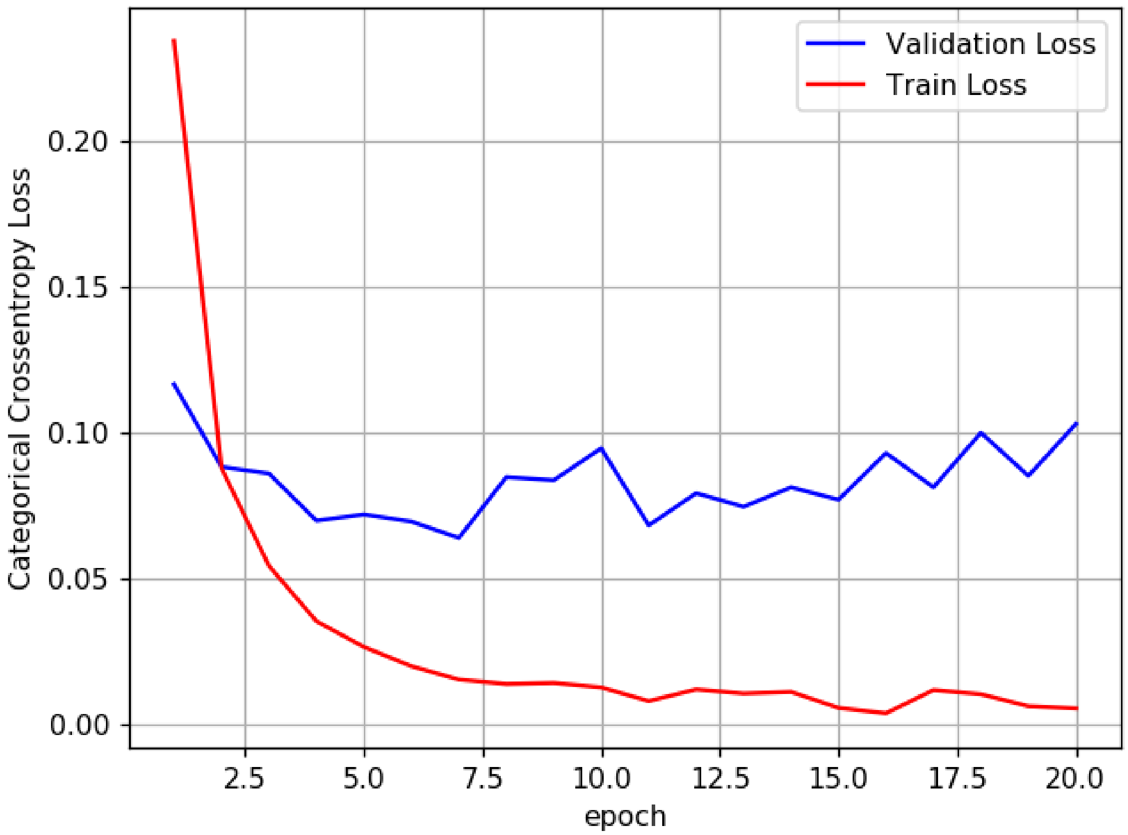
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_12 (Dense)             (None, 128)               65664
_________________________________________________________________
dense_13 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.2341 - ac
c: 0.9295 - val_loss: 0.1165 - val_acc: 0.9652
Epoch 2/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0878 - ac
c: 0.9729 - val_loss: 0.0883 - val_acc: 0.9720
Epoch 3/20
60000/60000 [==============================] - 5s 75us/step - loss: 0.0544 - ac
c: 0.9825 - val_loss: 0.0860 - val_acc: 0.9729
Epoch 4/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0354 - ac
c: 0.9885 - val_loss: 0.0699 - val_acc: 0.9797
Epoch 5/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0266 - ac
c: 0.9914 - val_loss: 0.0720 - val_acc: 0.9788
Epoch 6/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0200 - ac
c: 0.9941 - val_loss: 0.0696 - val_acc: 0.9803
Epoch 7/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0155 - ac
c: 0.9951 - val_loss: 0.0640 - val_acc: 0.9829
Epoch 8/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0140 - ac
c: 0.9952 - val_loss: 0.0848 - val_acc: 0.9792
Epoch 9/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0143 - ac
c: 0.9952 - val_loss: 0.0837 - val_acc: 0.9796
Epoch 10/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0128 - ac
c: 0.9958 - val_loss: 0.0946 - val_acc: 0.9782
Epoch 11/20
60000/60000 [==============================] - 7s 125us/step - loss: 0.0081 - ac
c: 0.9974 - val_loss: 0.0682 - val_acc: 0.9826
Epoch 12/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0121 - ac
c: 0.9959 - val_loss: 0.0793 - val_acc: 0.9816
Epoch 13/20
60000/60000 [==============================] - 8s 133us/step - loss: 0.0107 - ac
c: 0.9963 - val_loss: 0.0746 - val_acc: 0.9820
Epoch 14/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.0113 - ac
c: 0.9960 - val_loss: 0.0813 - val_acc: 0.9816
Epoch 15/20
60000/60000 [==============================] - 5s 77us/step - loss: 0.0058 - ac
```

```
c: 0.9982 - val_loss: 0.0770 - val_acc: 0.9842
Epoch 16/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0040 - ac
c: 0.9987 - val_loss: 0.0930 - val_acc: 0.9808
Epoch 17/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0119 - ac
c: 0.9959 - val_loss: 0.0813 - val_acc: 0.9819
Epoch 18/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0105 - ac
c: 0.9966 - val_loss: 0.1000 - val_acc: 0.9803
Epoch 19/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0064 - ac
c: 0.9981 - val_loss: 0.0852 - val_acc: 0.9831
Epoch 20/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.0056 - ac
c: 0.9982 - val_loss: 0.1029 - val_acc: 0.9805
```

In [0]:

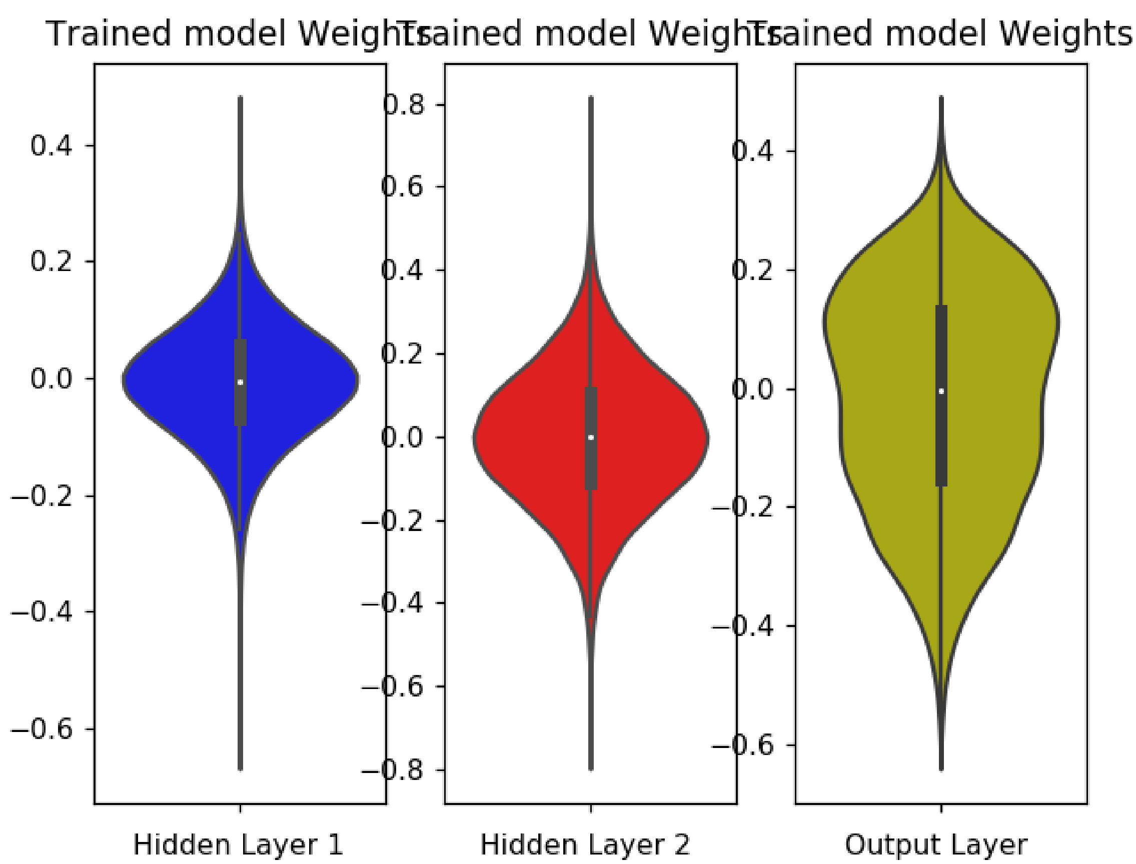
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.10294274219236926
Test accuracy: 0.9805
```



/

In [0]:
```
w_after = model_relu.get_weights()

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


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

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

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



MLP + Batch-Norm on hidden Layers + AdamOptimizer

In [24]:
```
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

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

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

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


model_batch.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_12 (Dense)             (None, 364)               285740    
_________________________________________________________________
batch_normalization_5 (Batch (None, 364)               1456      
_________________________________________________________________
dense_13 (Dense)             (None, 52)                18980     
_________________________________________________________________
batch_normalization_6 (Batch (None, 52)                208       
_________________________________________________________________
dense_14 (Dense)             (None, 10)                530       
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

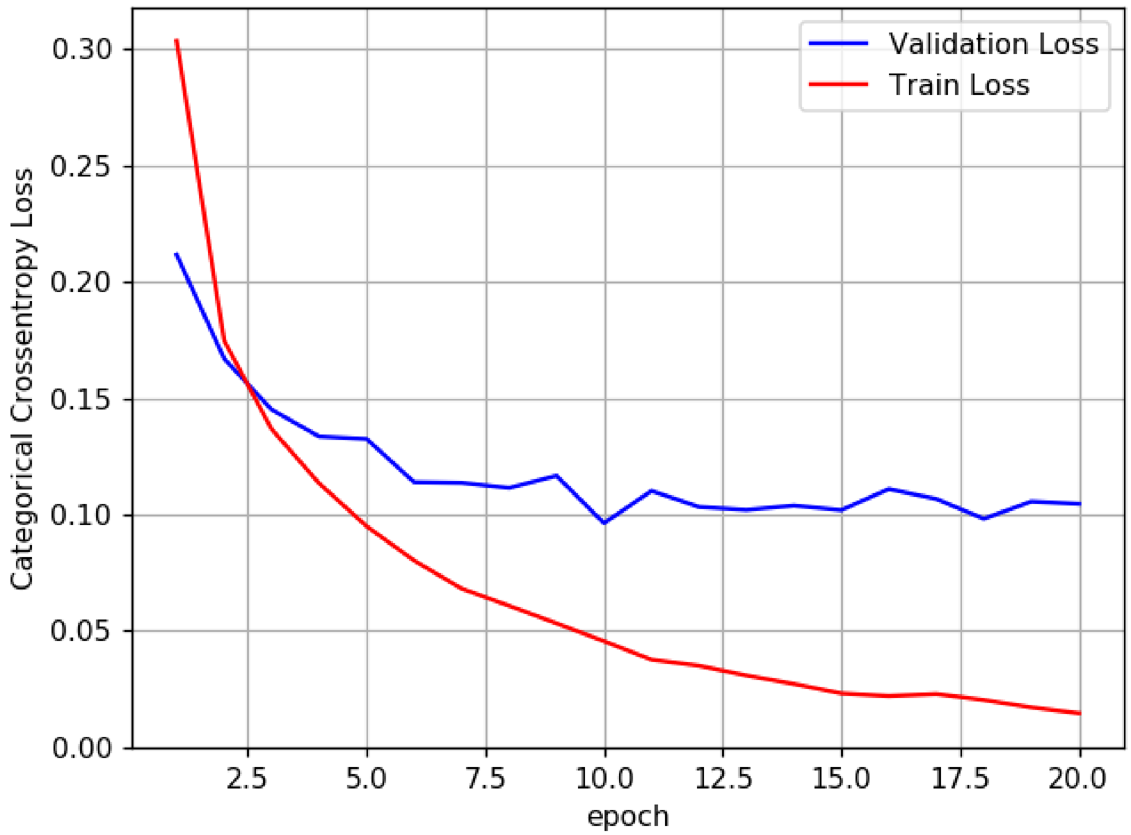
In [25]:
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=[
'accuracy'])

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoc
h, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 12s 208us/step - loss: 0.3459 - a
ccuracy: 0.8999 - val_loss: 0.2532 - val_accuracy: 0.9263
Epoch 2/20
60000/60000 [==============================] - 12s 202us/step - loss: 0.2152 - a
ccuracy: 0.9370 - val_loss: 0.2014 - val_accuracy: 0.9396
Epoch 3/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.1752 - a
ccuracy: 0.9481 - val_loss: 0.1796 - val_accuracy: 0.9464
Epoch 4/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.1470 - a
ccuracy: 0.9568 - val_loss: 0.1584 - val_accuracy: 0.9514
Epoch 5/20
60000/60000 [==============================] - 10s 168us/step - loss: 0.1272 - a
ccuracy: 0.9621 - val_loss: 0.1422 - val_accuracy: 0.9566
Epoch 6/20
60000/60000 [==============================] - 10s 167us/step - loss: 0.1096 - a
ccuracy: 0.9663 - val_loss: 0.1358 - val_accuracy: 0.9582
Epoch 7/20
60000/60000 [==============================] - 10s 170us/step - loss: 0.0944 - a
ccuracy: 0.9720 - val_loss: 0.1144 - val_accuracy: 0.9631
Epoch 8/20
60000/60000 [==============================] - 10s 161us/step - loss: 0.0797 - a
ccuracy: 0.9758 - val_loss: 0.1158 - val_accuracy: 0.9650
Epoch 9/20
60000/60000 [==============================] - 9s 148us/step - loss: 0.0684 - ac
curacy: 0.9788 - val_loss: 0.1073 - val_accuracy: 0.9661
Epoch 10/20
60000/60000 [==============================] - 10s 168us/step - loss: 0.0604 - a
ccuracy: 0.9812 - val_loss: 0.1053 - val_accuracy: 0.9687
Epoch 11/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.0505 - ac
curacy: 0.9846 - val_loss: 0.1008 - val_accuracy: 0.9689
Epoch 12/20
60000/60000 [==============================] - 10s 173us/step - loss: 0.0433 - a
ccuracy: 0.9862 - val_loss: 0.1016 - val_accuracy: 0.9690
Epoch 13/20
60000/60000 [==============================] - 12s 205us/step - loss: 0.0402 - a
ccuracy: 0.9871 - val_loss: 0.1003 - val_accuracy: 0.9718
Epoch 14/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.0342 - a
ccuracy: 0.9889 - val_loss: 0.1045 - val_accuracy: 0.9705
Epoch 15/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.0304 - a
ccuracy: 0.9905 - val_loss: 0.0907 - val_accuracy: 0.9739
Epoch 16/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.0274 - a
ccuracy: 0.9917 - val_loss: 0.0991 - val_accuracy: 0.9734
Epoch 17/20
60000/60000 [==============================] - 10s 169us/step - loss: 0.0251 - a
ccuracy: 0.9917 - val_loss: 0.1036 - val_accuracy: 0.9712
Epoch 18/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.0217 - a
ccuracy: 0.9931 - val_loss: 0.1055 - val_accuracy: 0.9712
Epoch 19/20
60000/60000 [==============================] - 12s 193us/step - loss: 0.0218 - a
ccuracy: 0.9928 - val_loss: 0.0967 - val_accuracy: 0.9733
Epoch 20/20
```

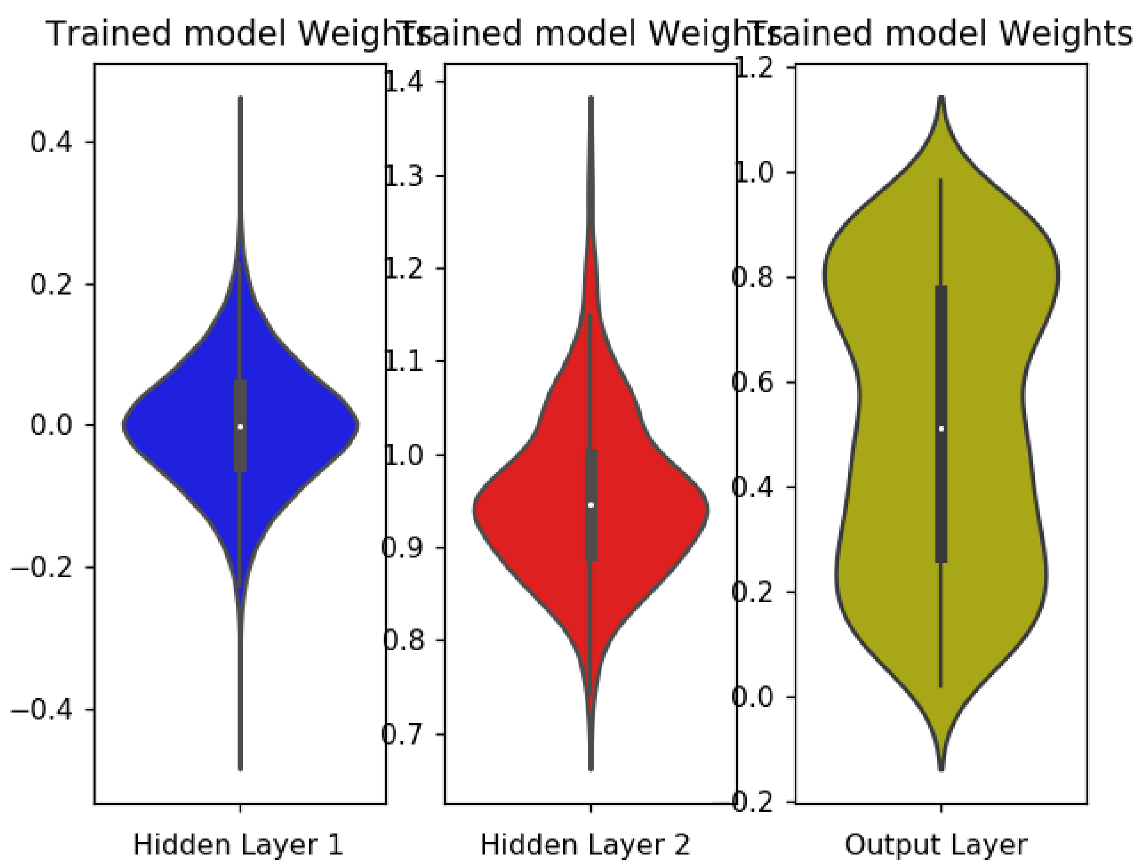
```
60000/60000 [==============================] - 11s 191us/step - loss: 0.0183 - a
ccuracy: 0.9942 - val_loss: 0.0927 - val_accuracy: 0.9770
```

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

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.10456635547156475
Test accuracy: 0.9732
```



In [0]:
```
w_after = model_batch.get_weights()

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


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

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

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



5. MLP + Dropout + AdamOptimizer

In [23]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batchnormaliza
tion-function-in-keras

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='sigmoid', 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(128, activation='sigmoid', kernel_initializer=RandomNormal(m
ean=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_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_9 (Dense)              (None, 512)               401920    
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_10 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_4 (Batch (None, 128)               512       
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [0]:

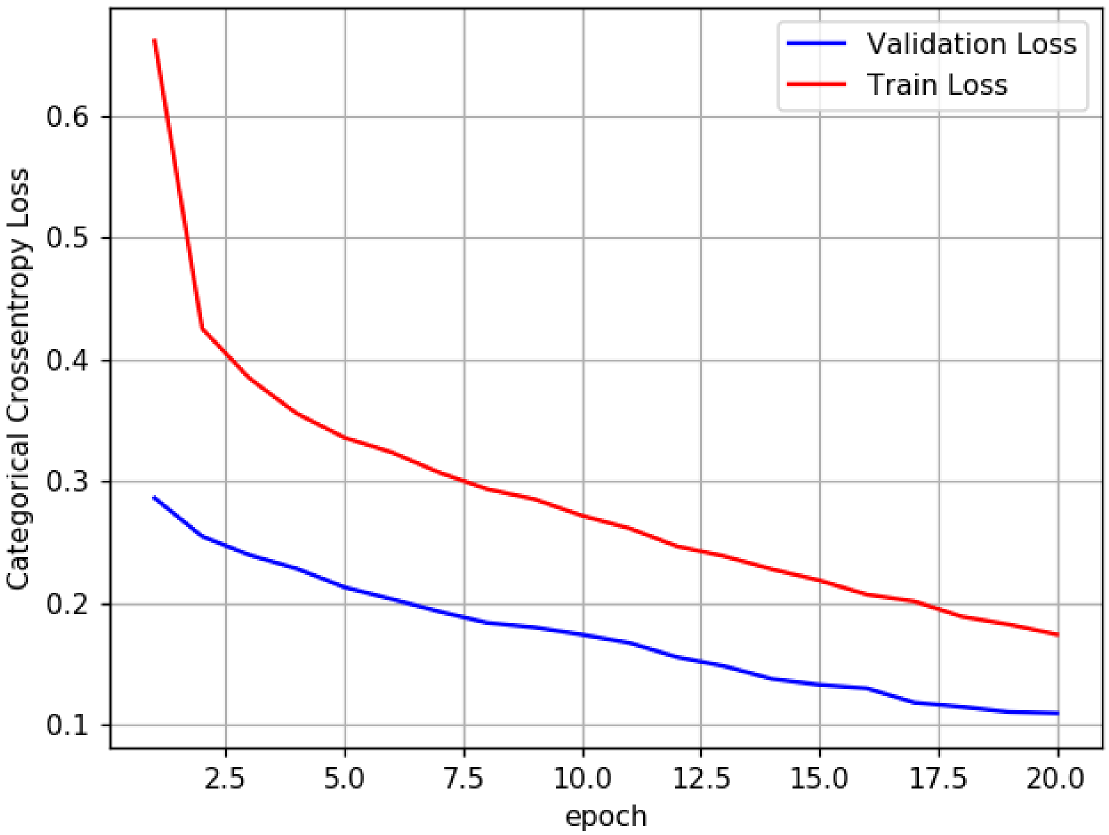
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a
ccuracy'])

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 [==============================] - 14s 227us/step - loss: 0.6612 - a
cc: 0.7951 - val_loss: 0.2860 - val_acc: 0.9166
Epoch 2/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.4250 - ac
c: 0.8710 - val_loss: 0.2545 - val_acc: 0.9252
Epoch 3/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.3841 - a
cc: 0.8846 - val_loss: 0.2391 - val_acc: 0.9298
Epoch 4/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.3551 - ac
c: 0.8927 - val_loss: 0.2279 - val_acc: 0.9325
Epoch 5/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.3355 - ac
c: 0.8986 - val_loss: 0.2127 - val_acc: 0.9356
Epoch 6/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.3234 - ac
c: 0.9031 - val_loss: 0.2029 - val_acc: 0.9387: 1s - loss:
Epoch 7/20
60000/60000 [==============================] - 8s 131us/step - loss: 0.3068 - ac
c: 0.9077 - val_loss: 0.1927 - val_acc: 0.9421
Epoch 8/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.2933 - a
cc: 0.9113 - val_loss: 0.1836 - val_acc: 0.9453
Epoch 9/20
60000/60000 [==============================] - 13s 222us/step - loss: 0.2850 - a
cc: 0.9131 - val_loss: 0.1797 - val_acc: 0.9451
Epoch 10/20
60000/60000 [==============================] - 14s 236us/step - loss: 0.2715 - a
cc: 0.9187 - val_loss: 0.1738 - val_acc: 0.9465
Epoch 11/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.2611 - ac
c: 0.9214 - val_loss: 0.1671 - val_acc: 0.9506
Epoch 12/20
60000/60000 [==============================] - 8s 134us/step - loss: 0.2464 - ac
c: 0.9252 - val_loss: 0.1554 - val_acc: 0.9525
Epoch 13/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.2382 - ac
c: 0.9278 - val_loss: 0.1479 - val_acc: 0.9554
Epoch 14/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.2275 - ac
c: 0.9313 - val_loss: 0.1375 - val_acc: 0.9580
Epoch 15/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.2183 - ac
c: 0.9337 - val_loss: 0.1326 - val_acc: 0.9599
Epoch 16/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.2068 - ac
c: 0.9384 - val_loss: 0.1297 - val_acc: 0.9613 loss: 0.2066 - ac
Epoch 17/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.2011 - ac
c: 0.9395 - val_loss: 0.1181 - val_acc: 0.9646
Epoch 18/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.1886 - ac
c: 0.9435 - val_loss: 0.1145 - val_acc: 0.9658
Epoch 19/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.1821 - ac
c: 0.9451 - val_loss: 0.1104 - val_acc: 0.9662
Epoch 20/20
```

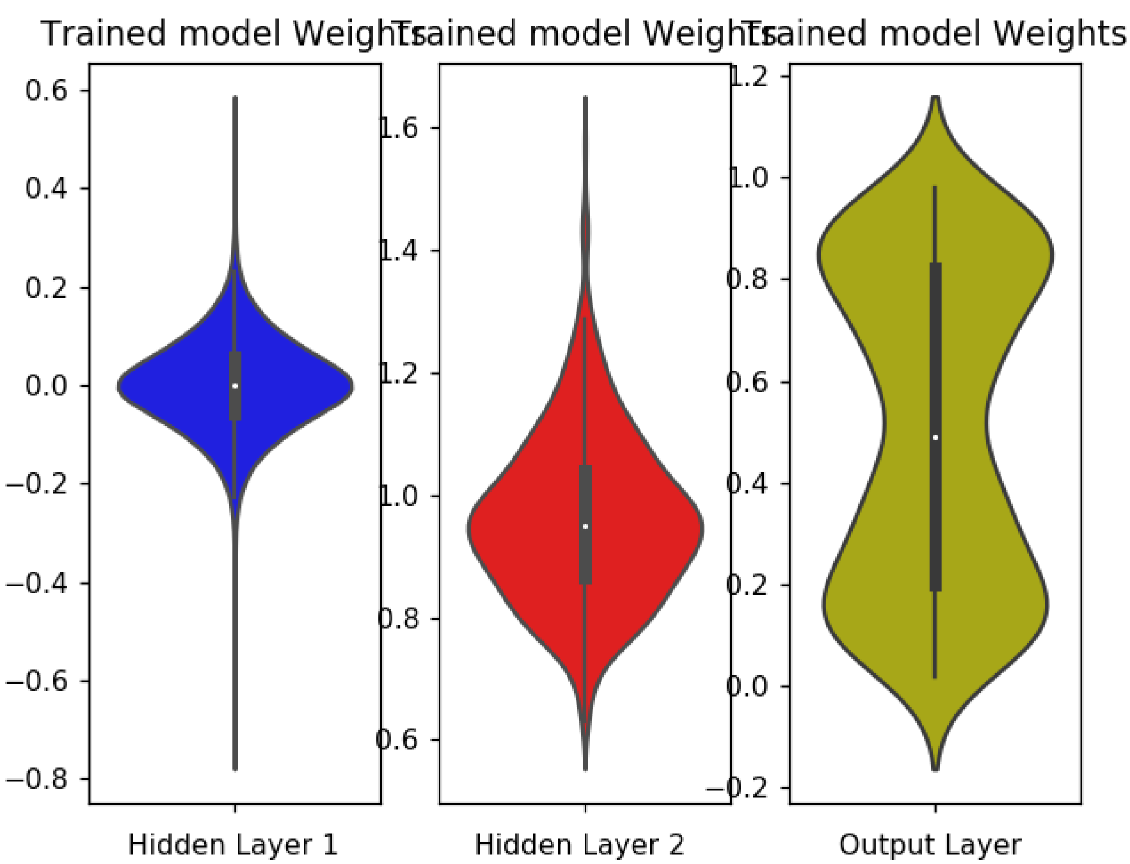
```
60000/60000 [==============================] - 8s 139us/step - loss: 0.1739 - ac
c: 0.9473 - val_loss: 0.1093 - val_acc: 0.9679
```

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

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.1093290721397847
Test accuracy: 0.9679
```



```
In [0]: w_after = model_drop.get_weights()

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


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

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

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



Hyper-parameter tuning of Keras models using Sklearn

```
In [0]: from keras.optimizers import Adam,RMSprop,SGD
        def best_hyperparameters(activ):

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


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

            return model
```

```
In [0]: # https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-mo
        dels-python-keras/

        activ = ['sigmoid','relu']

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

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

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

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

```
In [0]: print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
        means = grid_result.cv_results_['mean_test_score']
        stds = grid_result.cv_results_['std_test_score']
        params = grid_result.cv_results_['params']
        for mean, stdev, param in zip(means, stds, params):
            print("%f (%f) with: %r" % (mean, stdev, param))
```

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

Addtional Tries from Assignments

Model 1: MLP + Batch-Norm on hidden Layers + AdamOptimizer change dense of hidden layers </2>

In [28]:

```python
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(364, activation='sigmoid', input_shape=(input_dim,), kernel
_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(52, activation='sigmoid', kernel_initializer=RandomNormal(m
ean=0.0, stddev=0.55, seed=None)) )
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_15 (Dense)             (None, 364)               285740
_________________________________________________________________
batch_normalization_7 (Batch (None, 364)               1456
_________________________________________________________________
dense_16 (Dense)             (None, 52)                18980
_________________________________________________________________
batch_normalization_8 (Batch (None, 52)                208
_________________________________________________________________
dense_17 (Dense)             (None, 10)                530
=================================================================
Total params: 306,914
Trainable params: 306,082
Non-trainable params: 832
_________________________________________________________________
```

In [30]:

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

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoc
h, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.3523 - a
ccuracy: 0.8970 - val_loss: 0.2629 - val_accuracy: 0.9195
Epoch 2/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.2246 - ac
curacy: 0.9340 - val_loss: 0.2214 - val_accuracy: 0.9339 loss: 0.2247 - accurac
y: 0.
Epoch 3/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.1836 - ac
curacy: 0.9452 - val_loss: 0.1891 - val_accuracy: 0.9442
Epoch 4/20
60000/60000 [==============================] - 8s 135us/step - loss: 0.1511 - ac
curacy: 0.9553 - val_loss: 0.1742 - val_accuracy: 0.9493
Epoch 5/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.1325 - ac
curacy: 0.9612 - val_loss: 0.1527 - val_accuracy: 0.9564
Epoch 6/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.1122 - ac
curacy: 0.9664 - val_loss: 0.1412 - val_accuracy: 0.9576
Epoch 7/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.0967 - ac
curacy: 0.9706 - val_loss: 0.1268 - val_accuracy: 0.9637
Epoch 8/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.0832 - ac
curacy: 0.9747 - val_loss: 0.1251 - val_accuracy: 0.9635
Epoch 9/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0711 - ac
curacy: 0.9790 - val_loss: 0.1152 - val_accuracy: 0.9664
Epoch 10/20
60000/60000 [==============================] - 9s 155us/step - loss: 0.0609 - ac
curacy: 0.9814 - val_loss: 0.1119 - val_accuracy: 0.9683
Epoch 11/20
60000/60000 [==============================] - 8s 140us/step - loss: 0.0543 - ac
curacy: 0.9829 - val_loss: 0.1120 - val_accuracy: 0.9682
Epoch 12/20
60000/60000 [==============================] - 14s 237us/step - loss: 0.0455 - a
ccuracy: 0.9858 - val_loss: 0.1063 - val_accuracy: 0.9678
Epoch 13/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.0396 - a
ccuracy: 0.9873 - val_loss: 0.1053 - val_accuracy: 0.9715
Epoch 14/20
60000/60000 [==============================] - 13s 213us/step - loss: 0.0341 - a
ccuracy: 0.9895 - val_loss: 0.1029 - val_accuracy: 0.9707
Epoch 15/20
60000/60000 [==============================] - 10s 173us/step - loss: 0.0318 - a
ccuracy: 0.9899 - val_loss: 0.0992 - val_accuracy: 0.9726
Epoch 16/20
60000/60000 [==============================] - 10s 170us/step - loss: 0.0268 - a
ccuracy: 0.9915 - val_loss: 0.1039 - val_accuracy: 0.9732
Epoch 17/20
60000/60000 [==============================] - 10s 169us/step - loss: 0.0250 - a
ccuracy: 0.9924 - val_loss: 0.1096 - val_accuracy: 0.9718
Epoch 18/20
60000/60000 [==============================] - 10s 166us/step - loss: 0.0221 - a
ccuracy: 0.9931 - val_loss: 0.1113 - val_accuracy: 0.9713
Epoch 19/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.0205 - a
ccuracy: 0.9933 - val_loss: 0.1071 - val_accuracy: 0.9725
```

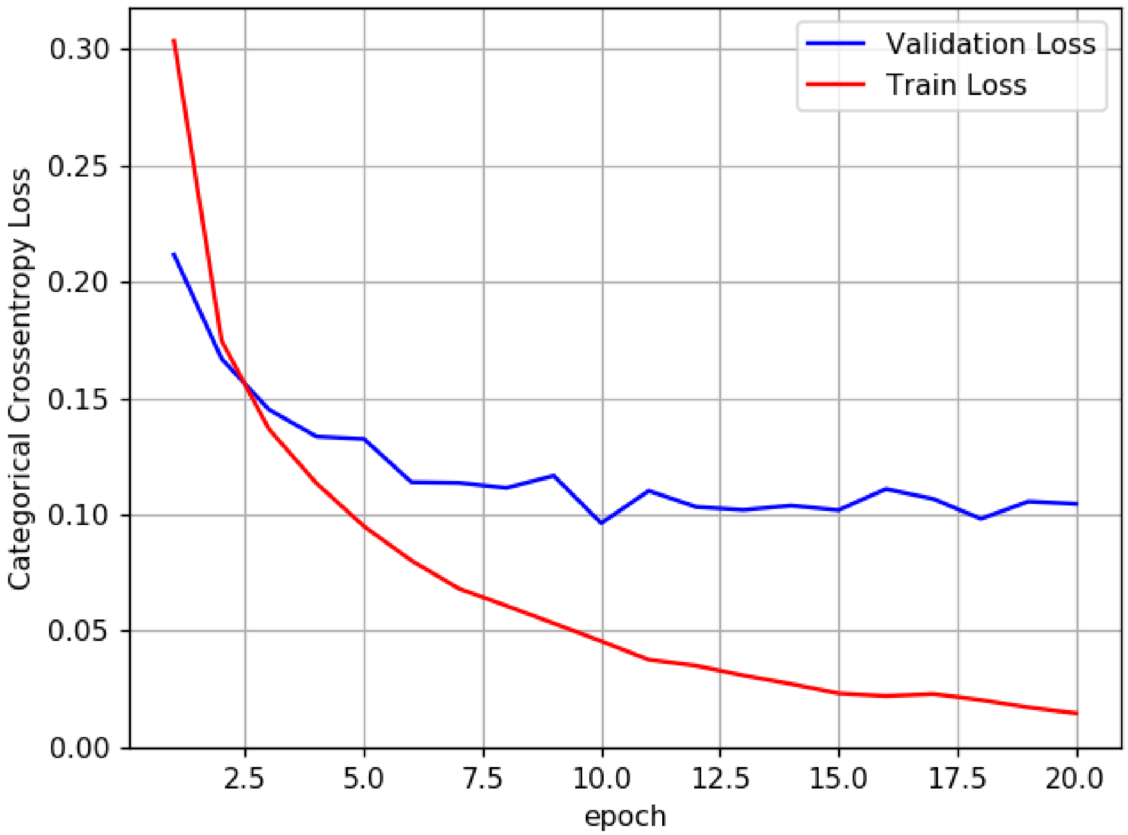
```
Epoch 20/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.0171 - a
ccuracy: 0.9948 - val_loss: 0.1001 - val_accuracy: 0.9752
```

Before (hidden 1 = 512, hidden 2 = 128)

In [0]:

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

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

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

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

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

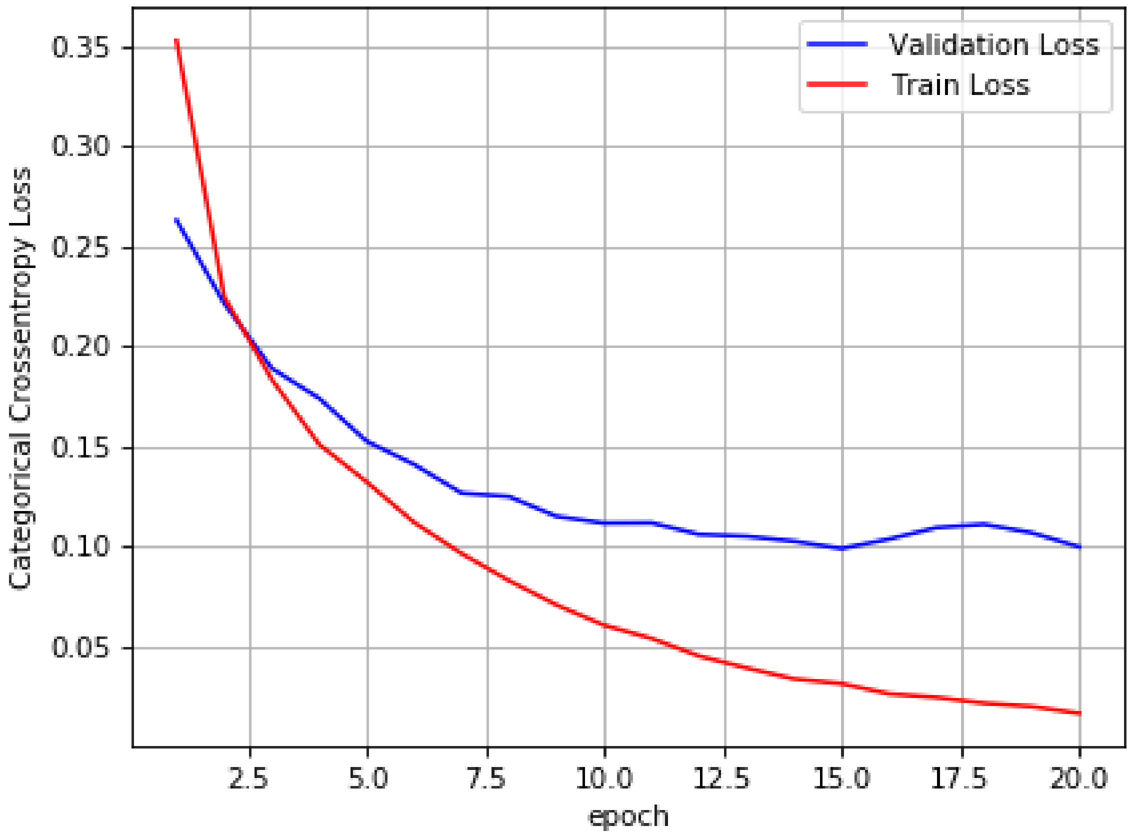
```
Test score: 0.10456635547156475
Test accuracy: 0.9732
```



After (hidden 1 = 364, hidden 2 = 52)

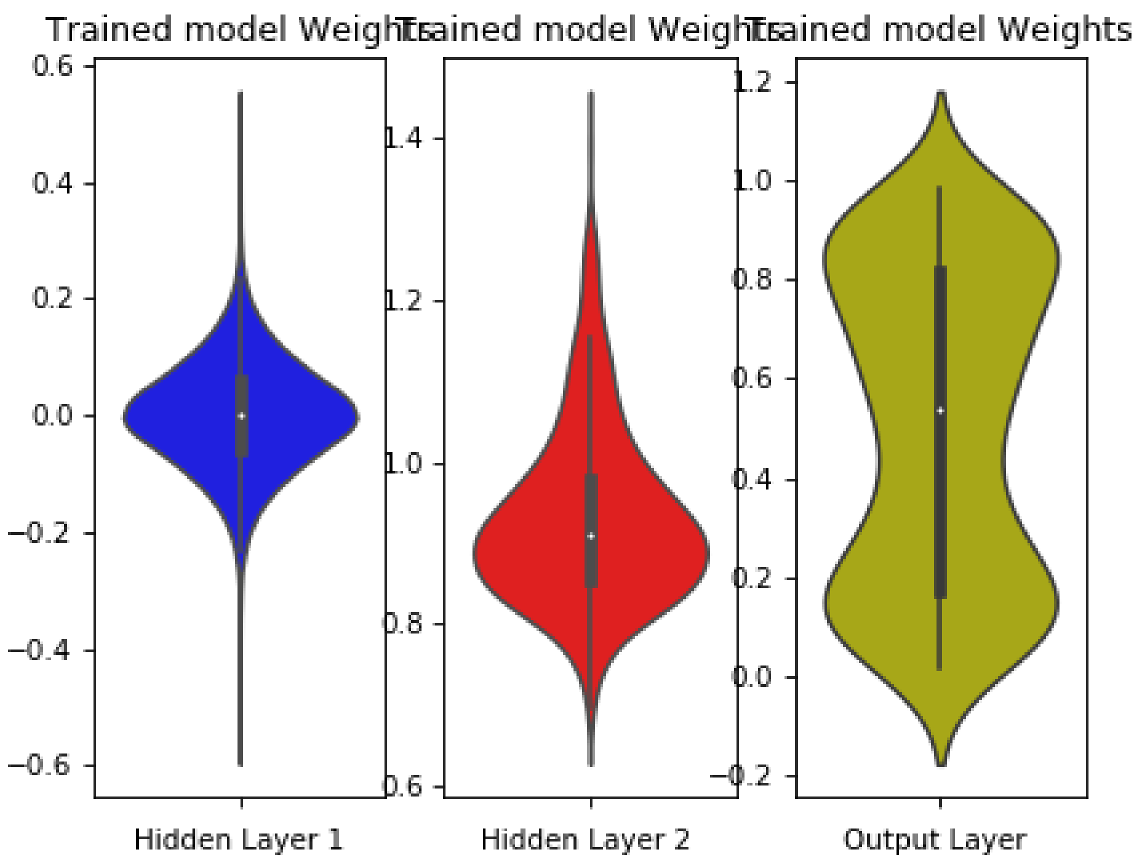
In [31]:
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.1000667967731366
Test accuracy: 0.9751999974250793
```



In [32]:

```
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()
```



conclusion for model 1: decrease activation units from 512 to 264 did not impact result much, where accuray or loss did not decrease

In [ ]:

```

```

Model 2: MLP + Batch-Norm on hidden Layers + AdamOptimizer Add + 3 hidden layer </2>

In [33]:
```
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

model_batch.add(Dense(364, activation='sigmoid', input_shape=(input_dim,), kernel
_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(128, activation='sigmoid', input_shape=(input_dim,), kernel
_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(52, activation='sigmoid', kernel_initializer=RandomNormal(m
ean=0.0, stddev=0.55, seed=None)) )
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_18 (Dense)             (None, 364)               285740
_________________________________________________________________
batch_normalization_9 (Batch (None, 364)               1456
_________________________________________________________________
dense_19 (Dense)             (None, 128)               46720
_________________________________________________________________
batch_normalization_10 (Batc (None, 128)               512
_________________________________________________________________
dense_20 (Dense)             (None, 52)                6708
_________________________________________________________________
batch_normalization_11 (Batc (None, 52)                208
_________________________________________________________________
dense_21 (Dense)             (None, 10)                530
=================================================================
Total params: 341,874
Trainable params: 340,786
Non-trainable params: 1,088
_________________________________________________________________
```

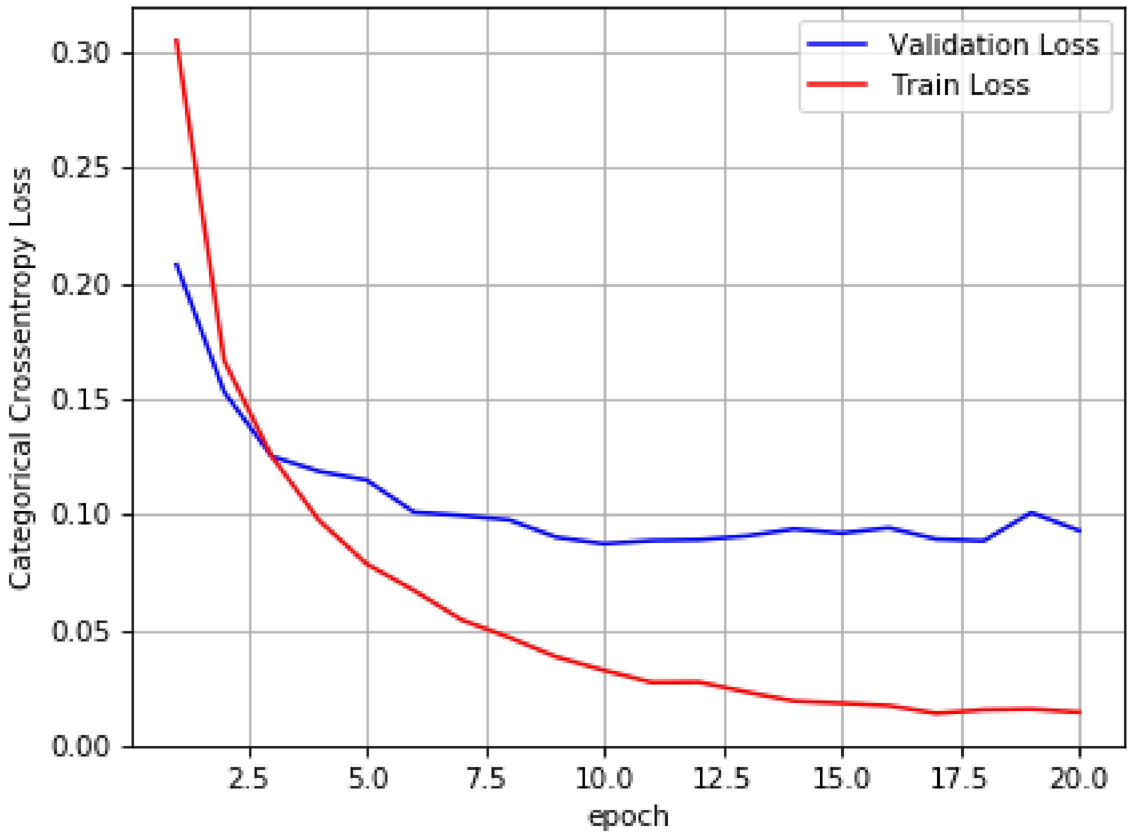
In [34]:
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=[
'accuracy'])

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoc
h, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 13s 213us/step - loss: 0.3049 - a
ccuracy: 0.9117 - val_loss: 0.2081 - val_accuracy: 0.9423
Epoch 2/20
60000/60000 [==============================] - 12s 192us/step - loss: 0.1666 - a
ccuracy: 0.9516 - val_loss: 0.1531 - val_accuracy: 0.9548
Epoch 3/20
60000/60000 [==============================] - 11s 189us/step - loss: 0.1254 - a
ccuracy: 0.9628 - val_loss: 0.1253 - val_accuracy: 0.9629
Epoch 4/20
60000/60000 [==============================] - 10s 164us/step - loss: 0.0976 - a
ccuracy: 0.9701 - val_loss: 0.1188 - val_accuracy: 0.9637
Epoch 5/20
60000/60000 [==============================] - 10s 168us/step - loss: 0.0788 - a
ccuracy: 0.9758 - val_loss: 0.1151 - val_accuracy: 0.9658
Epoch 6/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.0673 - a
ccuracy: 0.9791 - val_loss: 0.1012 - val_accuracy: 0.9703
Epoch 7/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0546 - a
ccuracy: 0.9834 - val_loss: 0.0997 - val_accuracy: 0.9684
Epoch 8/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.0470 - a
ccuracy: 0.9852 - val_loss: 0.0979 - val_accuracy: 0.9724
Epoch 9/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.0385 - a
ccuracy: 0.9874 - val_loss: 0.0903 - val_accuracy: 0.9730
Epoch 10/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.0328 - a
ccuracy: 0.9894 - val_loss: 0.0875 - val_accuracy: 0.9744
Epoch 11/20
60000/60000 [==============================] - 10s 173us/step - loss: 0.0275 - a
ccuracy: 0.9908 - val_loss: 0.0889 - val_accuracy: 0.9757
Epoch 12/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.0277 - a
ccuracy: 0.9911 - val_loss: 0.0892 - val_accuracy: 0.9745
Epoch 13/20
60000/60000 [==============================] - 11s 187us/step - loss: 0.0234 - a
ccuracy: 0.9926 - val_loss: 0.0909 - val_accuracy: 0.9740
Epoch 14/20
60000/60000 [==============================] - 10s 163us/step - loss: 0.0195 - a
ccuracy: 0.9936 - val_loss: 0.0939 - val_accuracy: 0.9750
Epoch 15/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0186 - a
ccuracy: 0.9940 - val_loss: 0.0921 - val_accuracy: 0.9760
Epoch 16/20
60000/60000 [==============================] - 11s 186us/step - loss: 0.0175 - a
ccuracy: 0.9943 - val_loss: 0.0944 - val_accuracy: 0.9759
Epoch 17/20
60000/60000 [==============================] - 10s 168us/step - loss: 0.0142 - a
ccuracy: 0.9953 - val_loss: 0.0895 - val_accuracy: 0.9763
Epoch 18/20
60000/60000 [==============================] - 11s 187us/step - loss: 0.0157 - a
ccuracy: 0.9950 - val_loss: 0.0888 - val_accuracy: 0.9772
Epoch 19/20
60000/60000 [==============================] - 12s 200us/step - loss: 0.0160 - a
ccuracy: 0.9947 - val_loss: 0.1010 - val_accuracy: 0.9740
Epoch 20/20
```

```
60000/60000 [==============================] - 12s 202us/step - loss: 0.0147 - a
ccuracy: 0.9952 - val_loss: 0.0932 - val_accuracy: 0.9765
```

In [35]:
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.0932293744858849
Test accuracy: 0.9764999747276306
```



conclusion for model 2: Adding another hidden layer slightly increased accuracy, and reduced log loss

```
In [ ]:
```

Model 3: MLP + Batch-Norm on hidden Layers + AdamOptimizer Add + 5 hidden layer </2>

In [38]:
```
# Multilayer perceptron

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

from keras.layers.normalization import BatchNormalization

model_batch = Sequential()

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


model_batch.add(Dense(364, activation='sigmoid', input_shape=(input_dim,), kernel
_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(128, activation='sigmoid', input_shape=(input_dim,), kernel
_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(52, activation='sigmoid', kernel_initializer=RandomNormal(m
ean=0.0, stddev=0.55, seed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(32, activation='sigmoid', input_shape=(input_dim,), kernel_
initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())


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


model_batch.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_22 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_12 (Batc (None, 512)               2048      
_________________________________________________________________
dense_23 (Dense)             (None, 364)               186732    
_________________________________________________________________
batch_normalization_13 (Batc (None, 364)               1456      
_________________________________________________________________
dense_24 (Dense)             (None, 128)               46720     
_________________________________________________________________
batch_normalization_14 (Batc (None, 128)               512       
_________________________________________________________________
dense_25 (Dense)             (None, 52)                6708      
_________________________________________________________________
batch_normalization_15 (Batc (None, 52)                208       
_________________________________________________________________
dense_26 (Dense)             (None, 32)                1696      
_________________________________________________________________
batch_normalization_16 (Batc (None, 32)                128       
_________________________________________________________________
dense_27 (Dense)             (None, 10)                330       
=================================================================
Total params: 648,458
Trainable params: 646,282
Non-trainable params: 2,176
_________________________________________________________________
```

In [39]:

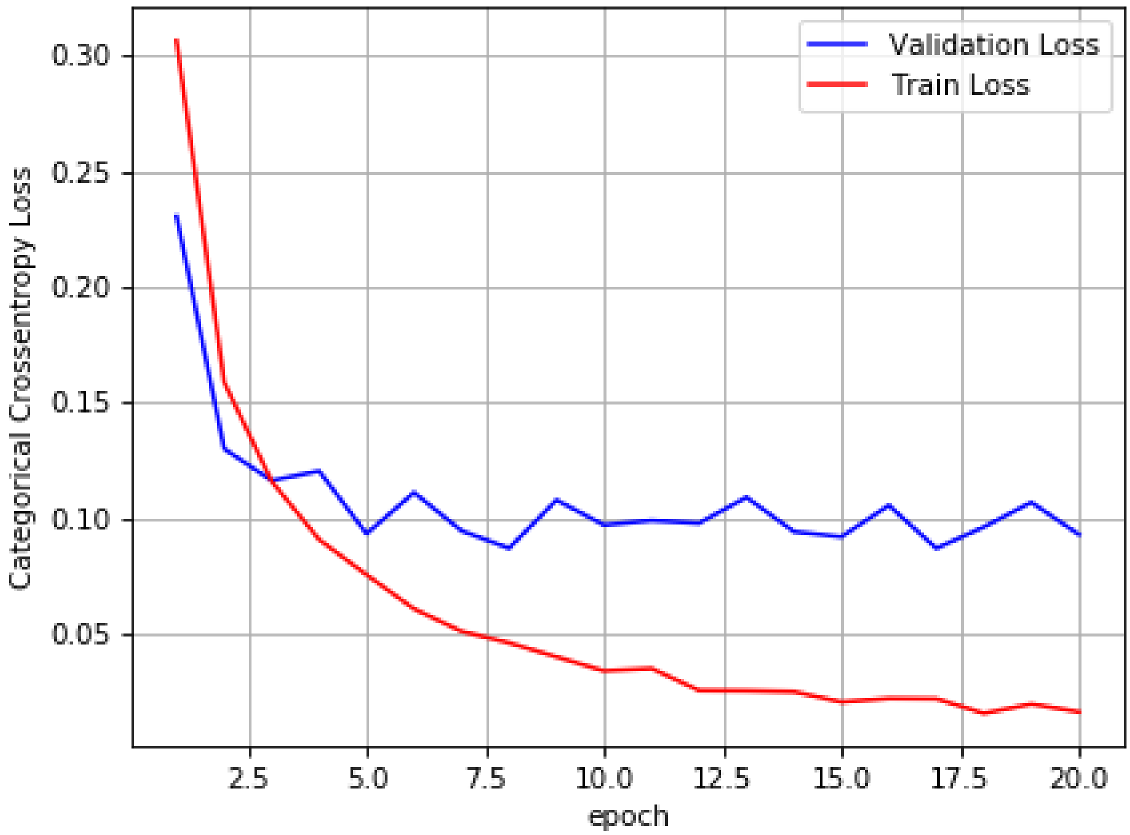
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=[
'accuracy'])

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoc
h, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 20s 328us/step - loss: 0.3063 - a
ccuracy: 0.9132 - val_loss: 0.2306 - val_accuracy: 0.9405
Epoch 2/20
60000/60000 [==============================] - 19s 318us/step - loss: 0.1589 - a
ccuracy: 0.9525 - val_loss: 0.1302 - val_accuracy: 0.9620
Epoch 3/20
60000/60000 [==============================] - 20s 331us/step - loss: 0.1162 - a
ccuracy: 0.9648 - val_loss: 0.1165 - val_accuracy: 0.9654
Epoch 4/20
60000/60000 [==============================] - 19s 320us/step - loss: 0.0910 - a
ccuracy: 0.9722 - val_loss: 0.1206 - val_accuracy: 0.9646
Epoch 5/20
60000/60000 [==============================] - 20s 329us/step - loss: 0.0758 - a
ccuracy: 0.9766 - val_loss: 0.0935 - val_accuracy: 0.9736
Epoch 6/20
60000/60000 [==============================] - 18s 300us/step - loss: 0.0611 - a
ccuracy: 0.9814 - val_loss: 0.1113 - val_accuracy: 0.9665
Epoch 7/20
60000/60000 [==============================] - 21s 356us/step - loss: 0.0514 - a
ccuracy: 0.9838 - val_loss: 0.0948 - val_accuracy: 0.9718
Epoch 8/20
60000/60000 [==============================] - 28s 466us/step - loss: 0.0464 - a
ccuracy: 0.9851 - val_loss: 0.0873 - val_accuracy: 0.9739
Epoch 9/20
60000/60000 [==============================] - 28s 464us/step - loss: 0.0403 - a
ccuracy: 0.9871 - val_loss: 0.1081 - val_accuracy: 0.9694
Epoch 10/20
60000/60000 [==============================] - 28s 466us/step - loss: 0.0343 - a
ccuracy: 0.9883 - val_loss: 0.0973 - val_accuracy: 0.9733
Epoch 11/20
60000/60000 [==============================] - 26s 429us/step - loss: 0.0353 - a
ccuracy: 0.9889 - val_loss: 0.0992 - val_accuracy: 0.9724
Epoch 12/20
60000/60000 [==============================] - 24s 400us/step - loss: 0.0259 - a
ccuracy: 0.9917 - val_loss: 0.0981 - val_accuracy: 0.9743
Epoch 13/20
60000/60000 [==============================] - 27s 450us/step - loss: 0.0257 - a
ccuracy: 0.9920 - val_loss: 0.1093 - val_accuracy: 0.9701
Epoch 14/20
60000/60000 [==============================] - 26s 427us/step - loss: 0.0253 - a
ccuracy: 0.9915 - val_loss: 0.0944 - val_accuracy: 0.9752
Epoch 15/20
60000/60000 [==============================] - 24s 395us/step - loss: 0.0209 - a
ccuracy: 0.9931 - val_loss: 0.0922 - val_accuracy: 0.9766
Epoch 16/20
60000/60000 [==============================] - 20s 328us/step - loss: 0.0225 - a
ccuracy: 0.9928 - val_loss: 0.1058 - val_accuracy: 0.9731
Epoch 17/20
60000/60000 [==============================] - 20s 335us/step - loss: 0.0223 - a
ccuracy: 0.9927 - val_loss: 0.0872 - val_accuracy: 0.9768
Epoch 18/20
60000/60000 [==============================] - 20s 339us/step - loss: 0.0161 - a
ccuracy: 0.9950 - val_loss: 0.0964 - val_accuracy: 0.9773
Epoch 19/20
60000/60000 [==============================] - 18s 305us/step - loss: 0.0200 - a
ccuracy: 0.9934 - val_loss: 0.1071 - val_accuracy: 0.9758
Epoch 20/20
```

```
60000/60000 [==============================] - 20s 332us/step - loss: 0.0167 - a
ccuracy: 0.9942 - val_loss: 0.0932 - val_accuracy: 0.9780
```

In [40]:
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

# we will get val_loss and val_acc only when you pass the paramter validation_dat
a
# 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: 0.09317940528480685
Test accuracy: 0.9779999852180481
```



conclusion for model 3: Adding two more hidden layer slightly increased accuracy, and reduced log loss

In [ ]: