Keras -- MLPs on MNIST

In [18]:

```
# 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 matplotlib.pyplot as plt
```

In [2]:

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
import time
# https://gist.github.com/greydanus/f6eee59eaf1d90fcb3b534a25
362cea4
# https://stackoverflow.com/a/14434334
# this function is used to update the plots for each epoch an
d 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 [20]:

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

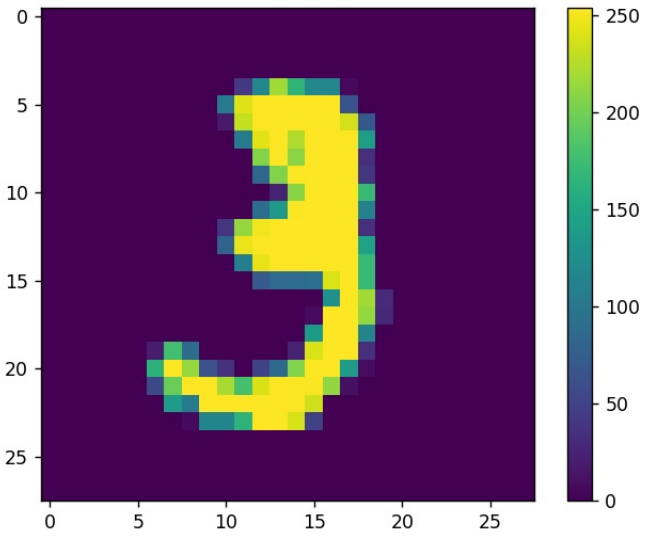
In [21]:

```
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 testing examples :", X_test.shape[0], "and e
ach image is of shape (%d, %d)"%(X_test.shape[1], X_test.shap
e[2]))
```

```
Number of training examples : 60000 and each i
mage is of shape (28, 28)
Number of testing examples : 10000 and each im
age is of shape (28, 28)
```

In [22]:

```
#viewing a sample data point from training
plt.figure()
plt.imshow(X_train[10])
plt.colorbar()
plt.grid(False)
plt.show()
```



In [23]:

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

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

In [24]:

```
# 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_te
st.shape[2])
```

In [25]:

```
# 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 testing examples :", X_test.shape[0], "and e
ach image is of shape (%d)"%(X_test.shape[1]))
```

```
Number of training examples : 60000 and each i
mage is of shape (784)
Number of testing examples : 10000 and each im
age is of shape (784)
```

In [26]:

```
X_train[0]
```

Out[26]:

```
array([0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
```

```
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 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, 0.0
7058824, 0.07058824,
       0.07058824, 0.49411765, 0.53333333, 0.6
8627451, 0.10196078,
       0.65098039, 1.        , 0.96862745, 0.4
9803922, 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.11764706, 0.14117647, 0.3
```

```
6862745, 0.60392157,
       0.66666667, 0.99215686, 0.99215686, 0.9
9215686, 0.99215686,
       0.99215686, 0.88235294, 0.6745098 , 0.9
9215686, 0.94901961,
       0.76470588, 0.25098039, 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.1
9215686, 0.93333333,
       0.99215686, 0.99215686, 0.99215686, 0.9
9215686, 0.99215686,
       0.99215686, 0.99215686, 0.99215686, 0.9
8431373, 0.36470588,
       0.32156863, 0.32156863, 0.21960784, 0.1
5294118, 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.07058824, 0.85882353, 0.9
9215686, 0.99215686,
       0.99215686, 0.99215686, 0.99215686, 0.7
7647059, 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, 0.9
9215686, 0.99215686,
       0.80392157, 0.04313725, 0.        , 0.1
6862745, 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, 0.3
5294118, 0.        ,
       0.        , 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, 0.7
4509804, 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, 0.8
8235294, 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, 0.9
9215686, 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, 0.5
8823529, 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.        , 0.0
627451 , 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.        , 0.9
7647059, 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.        , 0.1
8039216, 0.50980392,
       0.71764706, 0.99215686, 0.99215686, 0.8
1176471, 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, 0.9
```

```
9215686, 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, 0.9
9215686, 0.99215686,
       0.99215686, 0.99215686, 0.78823529, 0.3
0588235, 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.09019608, 0.25882353, 0.8
3529412, 0.99215686,
       0.99215686, 0.99215686, 0.99215686, 0.7
7647059, 0.31764706,
       0.00784314, 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.07058824, 0.6
7058824, 0.85882353,
       0.99215686, 0.99215686, 0.99215686, 0.9
9215686, 0.76470588,
       0.31372549, 0.03529412, 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
```

```
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.2
1568627, 0.6745098 ,
       0.88627451, 0.99215686, 0.99215686, 0.9
9215686, 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, 0.9
9215686, 0.99215686,
       0.83137255, 0.52941176, 0.51764706, 0.0
627451 , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
```

```
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       0.        , 0.        , 0.        , 0.
       , 0.        ,
       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 [27]:

```
# 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 [28]:

```
# 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 laye
r instances to the constructor:

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

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

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

###

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

# keras.layers.Dense(units, activation=None, use_bias=True, k
ernel_initializer='glorot_uniform',
# bias_initializer='zeros', kernel_regularizer=None, bias_reg
ularizer=None, activity_regularizer=None,
# kernel_constraint=None, bias_constraint=None)

# Dense implements the operation: output = activation(dot(inp
```

```
ut, kernel) + bias) where
# activation is the element-wise activation function passed a
s 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 = acti
vation(WT. X + b)

####

# https://keras.io/activations/

# Activations can either be used through an Activation layer,
 or through the activation 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 [29]:

```
# some model parameters
```

```
output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [30]:

```
# 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 automa
tic 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\Public\Anacon
da3\lib\site-packages\tensorflow\python\framew
ork\op_def_library.py:263: colocate_with (from
 tensorflow.python.framework.ops) is deprecate
d and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

In [31]:

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

```
# It receives three arguments:
# An optimizer. This could be the string identifier of an exi
sting 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=['accuracy'].  https://keras.io/m
etrics/


# Note: when using the categorical_crossentropy loss, your ta
rgets should be in categorical format
# (e.g. if you have 10 classes, the target for each sample sh
ould be a 10-dimensional vector that is all-zeros except
# for a 1 at the index corresponding to the class of the samp
le).

# that is why we converted out labels into vectors

model.compile(optimizer='sgd', 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 funct
ion

# fit(self, x=None, y=None, batch_size=None, epochs=1, verbos
e=1, callbacks=None, validation_split=0.0,
# validation_data=None, shuffle=True, class_weight=None, samp
le_weight=None, initial_epoch=0, steps_per_epoch=None,
# validation_steps=None)

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

# it returns A History object. Its History.history attribute
```

```
is a record of training loss values and
# metrics values at successive epochs, as well as validation
loss values and validation 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, verbose=1, validation_data=(X_test, Y_test))
```

```
WARNING:tensorflow:From C:\Users\Public\Anacon
da3\lib\site-packages\tensorflow\python\ops\ma
th_ops.py:3066: to_int32 (from tensorflow.pyth
on.ops.math_ops) is deprecated and will be rem
oved in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 1s 13us/step - loss: 1.2957 - acc: 0.6885 - v
al_loss: 0.8133 - val_acc: 0.8292
Epoch 2/20
60000/60000 [==============================] -
 1s 12us/step - loss: 0.7187 - acc: 0.8399 - v
al_loss: 0.6077 - val_acc: 0.8601
Epoch 3/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.5881 - acc: 0.8593 - v
al_loss: 0.5249 - val_acc: 0.8751
Epoch 4/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.5256 - acc: 0.8689 - v
al_loss: 0.4790 - val_acc: 0.8828
Epoch 5/20
60000/60000 [==============================] -
```

```
 1s 11us/step - loss: 0.4877 - acc: 0.8751 - v
al_loss: 0.4490 - val_acc: 0.8862
Epoch 6/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.4617 - acc: 0.8800 - v
al_loss: 0.4278 - val_acc: 0.8900
Epoch 7/20
60000/60000 [==============================] -
 1s 10us/step - loss: 0.4425 - acc: 0.8838 - v
al_loss: 0.4118 - val_acc: 0.8928
Epoch 8/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.4275 - acc: 0.8860 - v
al_loss: 0.3990 - val_acc: 0.8964
Epoch 9/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.4155 - acc: 0.8887 - v
al_loss: 0.3885 - val_acc: 0.8981
Epoch 10/20
60000/60000 [==============================] -
 1s 10us/step - loss: 0.4055 - acc: 0.8906 - v
al_loss: 0.3800 - val_acc: 0.9001
Epoch 11/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.3971 - acc: 0.8926 - v
al_loss: 0.3724 - val_acc: 0.9007
Epoch 12/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.3898 - acc: 0.8940 - v
al_loss: 0.3665 - val_acc: 0.9025
Epoch 13/20
60000/60000 [==============================] -
 1s 10us/step - loss: 0.3834 - acc: 0.8952 - v
al_loss: 0.3607 - val_acc: 0.9027
Epoch 14/20
60000/60000 [==============================] -
 1s 10us/step - loss: 0.3779 - acc: 0.8966 - v
```

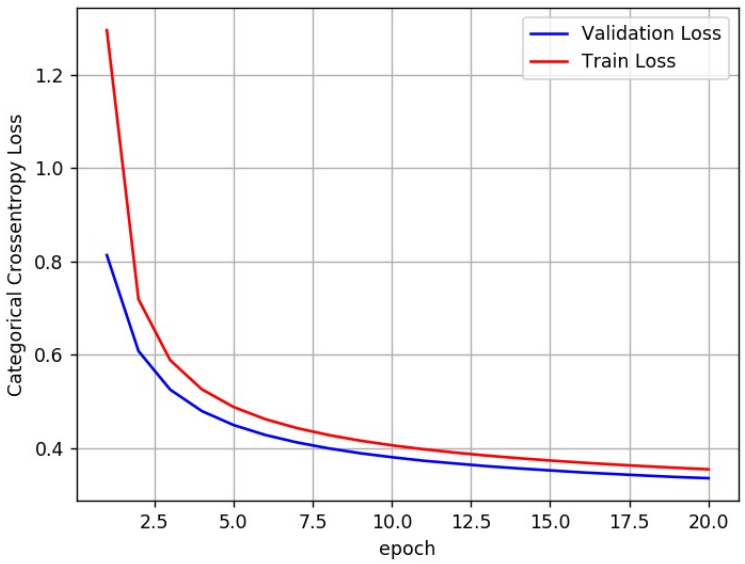
```
al_loss: 0.3560 - val_acc: 0.9039
Epoch 15/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.3729 - acc: 0.8978 - v
al_loss: 0.3517 - val_acc: 0.9050
Epoch 16/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.3684 - acc: 0.8985 - v
al_loss: 0.3474 - val_acc: 0.9058
Epoch 17/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.3643 - acc: 0.8997 - v
al_loss: 0.3440 - val_acc: 0.9066
Epoch 18/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.3606 - acc: 0.9005 - v
al_loss: 0.3405 - val_acc: 0.9073
Epoch 19/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.3572 - acc: 0.9011 - v
al_loss: 0.3374 - val_acc: 0.9084
Epoch 20/20
60000/60000 [==============================] -
 1s 11us/step - loss: 0.3540 - acc: 0.9019 - v
al_loss: 0.3350 - val_acc: 0.9094
```

In [32]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

```
Test score: 0.33496633795499803
Test accuracy: 0.9094
```


Validation Loss
Train Loss
Categorical Crossentropy Loss
1.2
1.0
0.8
0.6
0.4
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
epoch

architecture 1 -hidden layers 2: input (28 *28) hidden layer-1 (512) layer-2 (128) output softmax 10

MLP + Sigmoid activation + SGDOptimizer

In [33]:

```
# Multilayer perceptron

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

model_sigmoid.summary()
```

```
_________________________________________________
_____________________
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 [34]:

```
model_sigmoid.compile(optimizer='sgd', loss='categorical_cros
sentropy', metrics=['accuracy'])

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

```
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 3s 47us/step - loss: 2.2679 - acc: 0.2367 - v
al_loss: 2.2231 - val_acc: 0.3467
Epoch 2/20
60000/60000 [==============================] -
 3s 46us/step - loss: 2.1810 - acc: 0.4511 - v
al_loss: 2.1272 - val_acc: 0.5366
Epoch 3/20
60000/60000 [==============================] -
 3s 45us/step - loss: 2.0686 - acc: 0.5794 - v
al_loss: 1.9898 - val_acc: 0.6756
Epoch 4/20
60000/60000 [==============================] -
 3s 45us/step - loss: 1.9073 - acc: 0.6430 - v
al_loss: 1.7961 - val_acc: 0.7165
```

```
Epoch 5/20
60000/60000 [==============================] -
 3s 46us/step - loss: 1.6946 - acc: 0.6877 - v
al_loss: 1.5633 - val_acc: 0.7068
Epoch 6/20
60000/60000 [==============================] -
 3s 46us/step - loss: 1.4628 - acc: 0.7181 - v
al_loss: 1.3351 - val_acc: 0.7570
Epoch 7/20
60000/60000 [==============================] -
 3s 45us/step - loss: 1.2545 - acc: 0.7490 - v
al_loss: 1.1472 - val_acc: 0.7654
Epoch 8/20
60000/60000 [==============================] -
 3s 48us/step - loss: 1.0885 - acc: 0.7708 - v
al_loss: 1.0024 - val_acc: 0.7828
Epoch 9/20
60000/60000 [==============================] -
 3s 45us/step - loss: 0.9619 - acc: 0.7879 - v
al_loss: 0.8935 - val_acc: 0.8015
Epoch 10/20
60000/60000 [==============================] -
 3s 48us/step - loss: 0.8652 - acc: 0.8024 - v
al_loss: 0.8078 - val_acc: 0.8176
Epoch 11/20
60000/60000 [==============================] -
 3s 52us/step - loss: 0.7900 - acc: 0.8127 - v
al_loss: 0.7414 - val_acc: 0.8296
Epoch 12/20
60000/60000 [==============================] -
 3s 51us/step - loss: 0.7299 - acc: 0.8229 - v
al_loss: 0.6879 - val_acc: 0.8385
Epoch 13/20
60000/60000 [==============================] -
 3s 48us/step - loss: 0.6810 - acc: 0.8320 - v
al_loss: 0.6425 - val_acc: 0.8417
Epoch 14/20
```

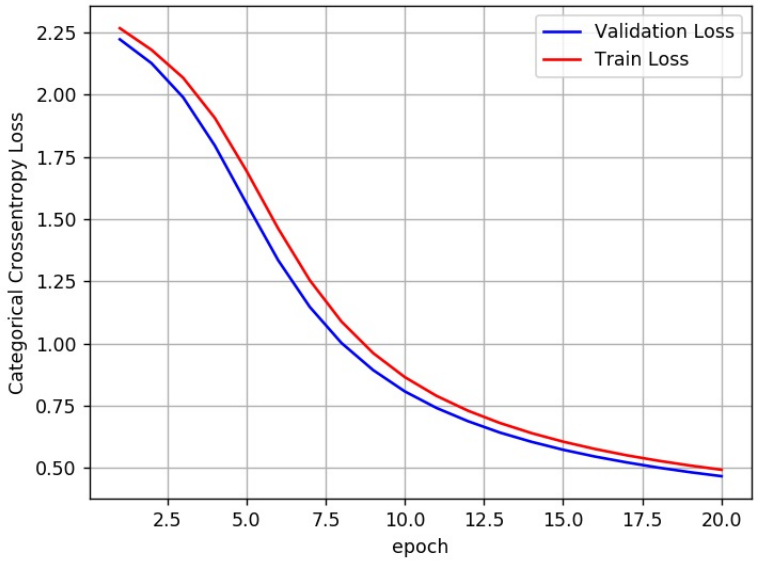
```
60000/60000 [==============================] -
 3s 51us/step - loss: 0.6402 - acc: 0.8399 - v
al_loss: 0.6053 - val_acc: 0.8486
Epoch 15/20
60000/60000 [==============================] -
 3s 51us/step - loss: 0.6058 - acc: 0.8465 - v
al_loss: 0.5733 - val_acc: 0.8554
Epoch 16/20
60000/60000 [==============================] -
 3s 47us/step - loss: 0.5765 - acc: 0.8526 - v
al_loss: 0.5461 - val_acc: 0.8604
Epoch 17/20
60000/60000 [==============================] -
 3s 49us/step - loss: 0.5512 - acc: 0.8577 - v
al_loss: 0.5225 - val_acc: 0.8669
Epoch 18/20
60000/60000 [==============================] -
 3s 49us/step - loss: 0.5292 - acc: 0.8628 - v
al_loss: 0.5016 - val_acc: 0.8715
Epoch 19/20
60000/60000 [==============================] -
 3s 51us/step - loss: 0.5099 - acc: 0.8667 - v
al_loss: 0.4832 - val_acc: 0.8732
Epoch 20/20
60000/60000 [==============================] -
 3s 50us/step - loss: 0.4929 - acc: 0.8705 - v
al_loss: 0.4670 - val_acc: 0.8783
```

In [35]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

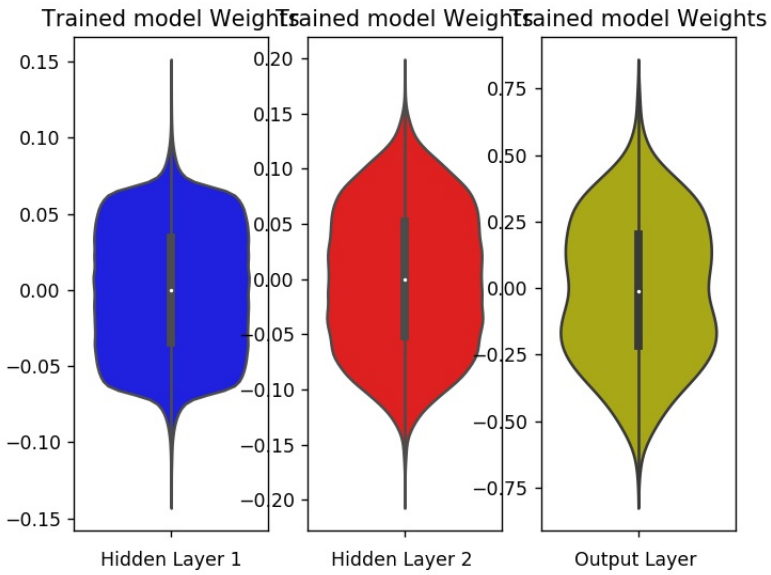
```
Test score: 0.4670006361246109
Test accuracy: 0.8783
```



In [36]:

```
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 [37]:

```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shap
e=(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_cro
ssentropy', metrics=['accuracy'])

history = model_sigmoid.fit(X_train, Y_train, batch_size=batc
h_size, epochs=nb_epoch, 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 samp
les
Epoch 1/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.5363 - acc: 0.8599 - v
al_loss: 0.2539 - val_acc: 0.9261
Epoch 2/20
60000/60000 [==============================] -
 4s 70us/step - loss: 0.2203 - acc: 0.9352 - v
al_loss: 0.1913 - val_acc: 0.9430
Epoch 3/20
60000/60000 [==============================] -
 5s 76us/step - loss: 0.1632 - acc: 0.9516 - v
al_loss: 0.1449 - val_acc: 0.9560
Epoch 4/20
60000/60000 [==============================] -
 4s 73us/step - loss: 0.1270 - acc: 0.9622 - v
al_loss: 0.1220 - val_acc: 0.9630
Epoch 5/20
60000/60000 [==============================] -
 5s 81us/step - loss: 0.1012 - acc: 0.9703 - v
al_loss: 0.1126 - val_acc: 0.9652
Epoch 6/20
60000/60000 [==============================] -
 4s 75us/step - loss: 0.0813 - acc: 0.9759 - v
al_loss: 0.0923 - val_acc: 0.9704
Epoch 7/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0659 - acc: 0.9804 - v
```

```
al_loss: 0.0855 - val_acc: 0.9737
Epoch 8/20
60000/60000 [==============================] -
 5s 78us/step - loss: 0.0536 - acc: 0.9844 - v
al_loss: 0.0751 - val_acc: 0.9769
Epoch 9/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0434 - acc: 0.9874 - v
al_loss: 0.0702 - val_acc: 0.9786
Epoch 10/20
60000/60000 [==============================] -
 5s 80us/step - loss: 0.0352 - acc: 0.9899 - v
al_loss: 0.0681 - val_acc: 0.9788
Epoch 11/20
60000/60000 [==============================] -
 5s 75us/step - loss: 0.0281 - acc: 0.9927 - v
al_loss: 0.0635 - val_acc: 0.9800
Epoch 12/20
60000/60000 [==============================] -
 5s 81us/step - loss: 0.0230 - acc: 0.9940 - v
al_loss: 0.0703 - val_acc: 0.9781
Epoch 13/20
60000/60000 [==============================] -
 5s 75us/step - loss: 0.0178 - acc: 0.9954 - v
al_loss: 0.0759 - val_acc: 0.9771
Epoch 14/20
60000/60000 [==============================] -
 5s 77us/step - loss: 0.0147 - acc: 0.9963 - v
al_loss: 0.0636 - val_acc: 0.9806
Epoch 15/20
60000/60000 [==============================] -
 4s 75us/step - loss: 0.0105 - acc: 0.9977 - v
al_loss: 0.0671 - val_acc: 0.9802
Epoch 16/20
60000/60000 [==============================] -
 4s 73us/step - loss: 0.0085 - acc: 0.9982 - v
al_loss: 0.0641 - val_acc: 0.9826
```

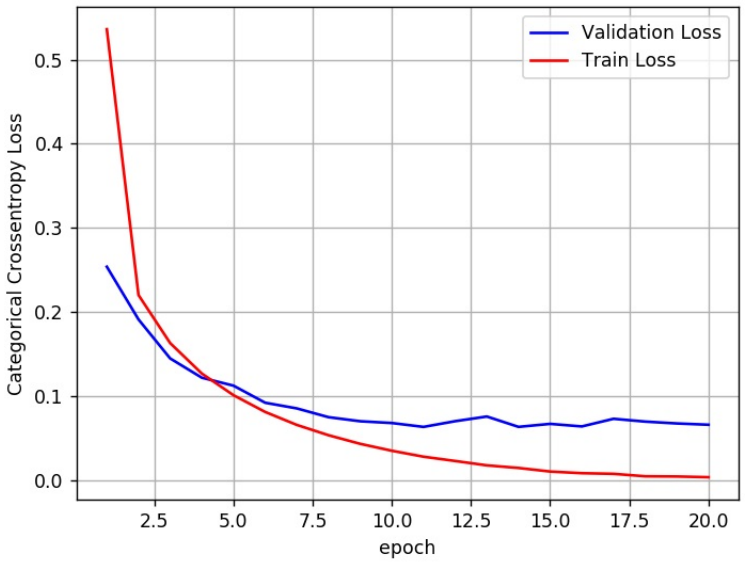
```
Epoch 17/20
60000/60000 [==============================] -
 5s 76us/step - loss: 0.0078 - acc: 0.9983 - v
al_loss: 0.0732 - val_acc: 0.9798
Epoch 18/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0048 - acc: 0.9992 - v
al_loss: 0.0698 - val_acc: 0.9820
Epoch 19/20
60000/60000 [==============================] -
 4s 70us/step - loss: 0.0046 - acc: 0.9990 - v
al_loss: 0.0676 - val_acc: 0.9827
Epoch 20/20
60000/60000 [==============================] -
 4s 75us/step - loss: 0.0037 - acc: 0.9994 - v
al_loss: 0.0661 - val_acc: 0.9828
```

In [38]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
```

```
amter 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 le
ngth equal to number of epochs

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

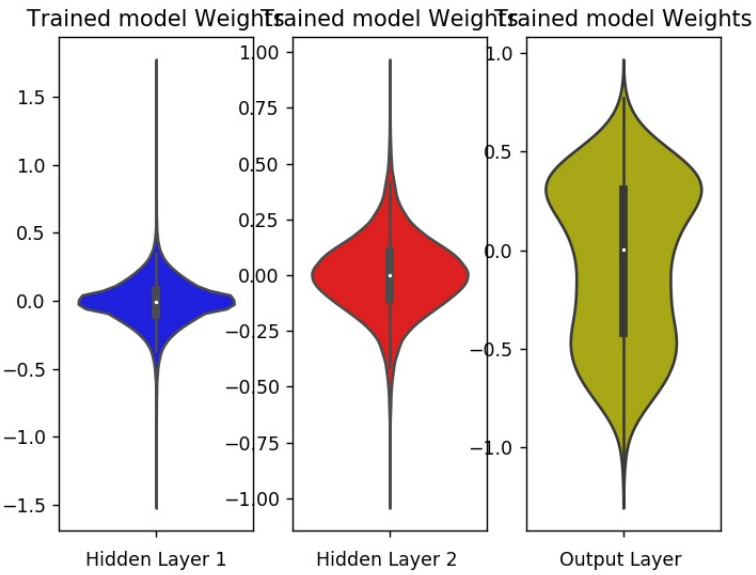
```
Test score: 0.06611316717128793
Test accuracy: 0.9828
```



In [39]:

```
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 + ReLU +SGD

In [40]:

```
# 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 s
atisfy this condition 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=(inp
ut_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.
062, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializ
er=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 [41]:

```
model_relu.compile(optimizer='sgd', loss='categorical_crossen
tropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_s
ize, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_t
est))
```

```
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 3s 49us/step - loss: 0.7412 - acc: 0.7898 - v
al_loss: 0.3874 - val_acc: 0.8935
Epoch 2/20
60000/60000 [==============================] -
 3s 46us/step - loss: 0.3513 - acc: 0.9013 - v
al_loss: 0.3035 - val_acc: 0.9132
Epoch 3/20
60000/60000 [==============================] -
 3s 47us/step - loss: 0.2913 - acc: 0.9169 - v
al_loss: 0.2635 - val_acc: 0.9237
Epoch 4/20
```

```
60000/60000 [==============================] -
 3s 47us/step - loss: 0.2577 - acc: 0.9265 - v
al_loss: 0.2407 - val_acc: 0.9316
Epoch 5/20
60000/60000 [==============================] -
 3s 46us/step - loss: 0.2345 - acc: 0.9335 - v
al_loss: 0.2209 - val_acc: 0.9377
Epoch 6/20
60000/60000 [==============================] -
 3s 46us/step - loss: 0.2161 - acc: 0.9388 - v
al_loss: 0.2079 - val_acc: 0.9421
Epoch 7/20
60000/60000 [==============================] -
 3s 47us/step - loss: 0.2012 - acc: 0.9433 - v
al_loss: 0.1945 - val_acc: 0.9455
Epoch 8/20
60000/60000 [==============================] -
 3s 47us/step - loss: 0.1885 - acc: 0.9469 - v
al_loss: 0.1870 - val_acc: 0.9475
Epoch 9/20
60000/60000 [==============================] -
 3s 46us/step - loss: 0.1775 - acc: 0.9502 - v
al_loss: 0.1769 - val_acc: 0.9507
Epoch 10/20
60000/60000 [==============================] -
 3s 48us/step - loss: 0.1679 - acc: 0.9528 - v
al_loss: 0.1686 - val_acc: 0.9519
Epoch 11/20
60000/60000 [==============================] -
 3s 51us/step - loss: 0.1594 - acc: 0.9552 - v
al_loss: 0.1632 - val_acc: 0.9531
Epoch 12/20
60000/60000 [==============================] -
 3s 46us/step - loss: 0.1517 - acc: 0.9579 - v
al_loss: 0.1561 - val_acc: 0.9551
Epoch 13/20
60000/60000 [==============================] -
```

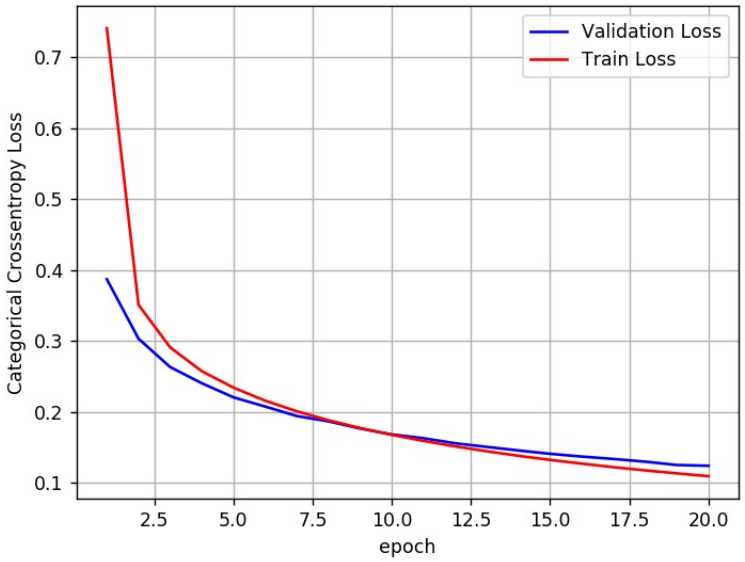
```
 3s 52us/step - loss: 0.1448 - acc: 0.9595 - v
al_loss: 0.1510 - val_acc: 0.9565
Epoch 14/20
60000/60000 [==============================] -
 3s 47us/step - loss: 0.1384 - acc: 0.9616 - v
al_loss: 0.1460 - val_acc: 0.9567
Epoch 15/20
60000/60000 [==============================] -
 3s 52us/step - loss: 0.1327 - acc: 0.9630 - v
al_loss: 0.1412 - val_acc: 0.9580
Epoch 16/20
60000/60000 [==============================] -
 3s 52us/step - loss: 0.1274 - acc: 0.9646 - v
al_loss: 0.1374 - val_acc: 0.9597
Epoch 17/20
60000/60000 [==============================] -
 3s 47us/step - loss: 0.1224 - acc: 0.9660 - v
al_loss: 0.1340 - val_acc: 0.9613
Epoch 18/20
60000/60000 [==============================] -
 3s 49us/step - loss: 0.1179 - acc: 0.9675 - v
al_loss: 0.1302 - val_acc: 0.9617
Epoch 19/20
60000/60000 [==============================] -
 3s 56us/step - loss: 0.1137 - acc: 0.9689 - v
al_loss: 0.1255 - val_acc: 0.9625
Epoch 20/20
60000/60000 [==============================] -
 3s 50us/step - loss: 0.1097 - acc: 0.9699 - v
al_loss: 0.1244 - val_acc: 0.9630
```

In [42]:

```
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, validation_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)
```

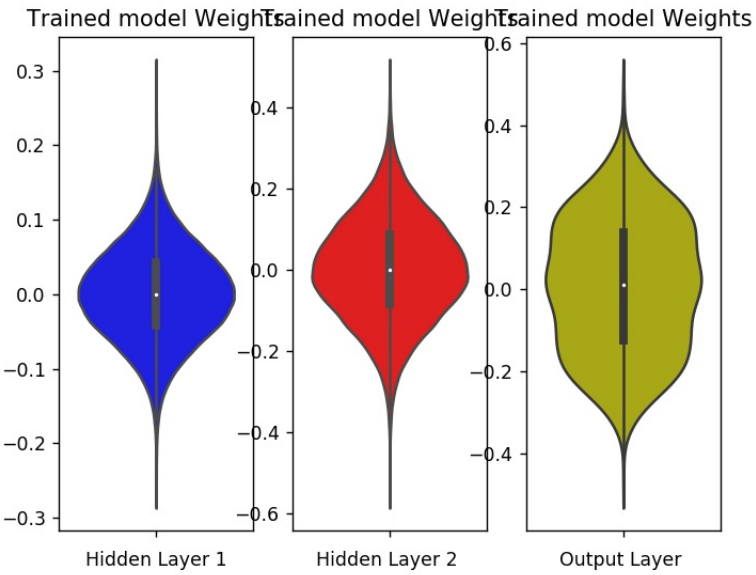
```
Test score: 0.12436589284352959
Test accuracy: 0.963
```



In [43]:

```
w_after = model_relu.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[2].flatten().reshape(-1,1)
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 [44]:

```
model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(inp
ut_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.
062, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializ
er=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_crosse
ntropy', metrics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_s
ize, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_t
est))
```

```
_________________________________________________________
_________________
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 samp
les
Epoch 1/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.2338 - acc: 0.9307 - v
al_loss: 0.1107 - val_acc: 0.9669
Epoch 2/20
60000/60000 [==============================] -
 4s 68us/step - loss: 0.0876 - acc: 0.9735 - v
al_loss: 0.0840 - val_acc: 0.9726
Epoch 3/20
60000/60000 [==============================] -
 4s 75us/step - loss: 0.0562 - acc: 0.9826 - v
al_loss: 0.0803 - val_acc: 0.9756
Epoch 4/20
60000/60000 [==============================] -
 4s 73us/step - loss: 0.0385 - acc: 0.9880 - v
al_loss: 0.0762 - val_acc: 0.9771
Epoch 5/20
60000/60000 [==============================] -
 4s 73us/step - loss: 0.0262 - acc: 0.9921 - v
al_loss: 0.0875 - val_acc: 0.9733
Epoch 6/20
60000/60000 [==============================] -
 5s 75us/step - loss: 0.0199 - acc: 0.9936 - v
al_loss: 0.0745 - val_acc: 0.9771
```

```
Epoch 7/20
60000/60000 [==============================] -
 4s 73us/step - loss: 0.0179 - acc: 0.9942 - v
al_loss: 0.0926 - val_acc: 0.9744
Epoch 8/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0158 - acc: 0.9949 - v
al_loss: 0.0781 - val_acc: 0.9798
Epoch 9/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0131 - acc: 0.9955 - v
al_loss: 0.0934 - val_acc: 0.9766
Epoch 10/20
60000/60000 [==============================] -
 5s 80us/step - loss: 0.0106 - acc: 0.9966 - v
al_loss: 0.0751 - val_acc: 0.9809
Epoch 11/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0117 - acc: 0.9962 - v
al_loss: 0.0722 - val_acc: 0.9811
Epoch 12/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0085 - acc: 0.9973 - v
al_loss: 0.0893 - val_acc: 0.9799
Epoch 13/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0117 - acc: 0.9963 - v
al_loss: 0.0819 - val_acc: 0.9800
Epoch 14/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0103 - acc: 0.9967 - v
al_loss: 0.0917 - val_acc: 0.9795
Epoch 15/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0099 - acc: 0.9968 - v
al_loss: 0.0864 - val_acc: 0.9808
Epoch 16/20
```

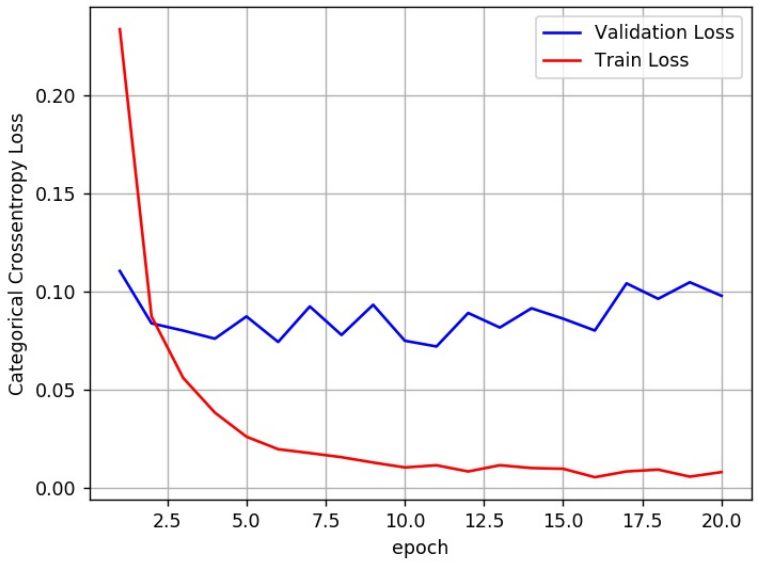
```
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0056 - acc: 0.9981 - v
al_loss: 0.0803 - val_acc: 0.9832
Epoch 17/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0085 - acc: 0.9974 - v
al_loss: 0.1044 - val_acc: 0.9797
Epoch 18/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0095 - acc: 0.9970 - v
al_loss: 0.0965 - val_acc: 0.9797
Epoch 19/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0059 - acc: 0.9982 - v
al_loss: 0.1049 - val_acc: 0.9802
Epoch 20/20
60000/60000 [==============================] -
 4s 74us/step - loss: 0.0082 - acc: 0.9971 - v
al_loss: 0.0980 - val_acc: 0.9810
```

In [45]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
```

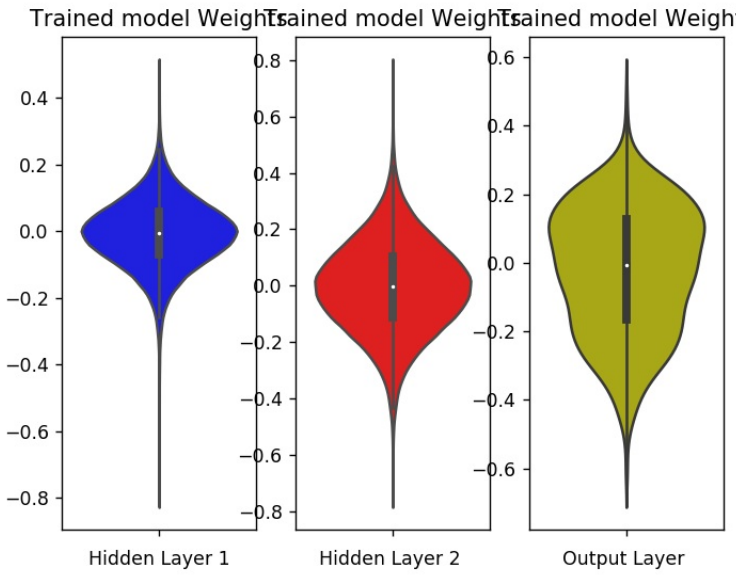
```
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs


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

```
Test score: 0.09800619804806311
Test accuracy: 0.981
```



In [46]:

```
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()
```


Trained model Weights
Trained model Weights
Trained model Weights
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Hidden Layer 1
Hidden Layer 2
Output Layer

MLP + Batch-Norm on hidden Layers + AdamOptimizer </2>

In [47]:

```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we s
atisfy this condition 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, stdde
v=0.039, seed=None)))
model_batch.add(BatchNormalization())

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

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


model_batch.summary()
```

_____

```
_________________
Layer (type)                 Output Shape
         Param #
=================================================
==================
dense_14 (Dense)             (None, 512)
         401920
_________________________________________________
_________________
batch_normalization_1 (Batch (None, 512)
         2048
_________________________________________________
_________________
dense_15 (Dense)             (None, 128)
         65664
_________________________________________________
_________________
batch_normalization_2 (Batch (None, 128)
         512
_________________________________________________
_________________
dense_16 (Dense)             (None, 10)
         1290
=================================================
==================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________
_________________
```

In [48]:

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

history = model_batch.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 samp
les
Epoch 1/20
60000/60000 [==============================] -
 5s 89us/step - loss: 0.3067 - acc: 0.9107 - v
al_loss: 0.2121 - val_acc: 0.9393
Epoch 2/20
60000/60000 [==============================] -
 5s 78us/step - loss: 0.1769 - acc: 0.9475 - v
al_loss: 0.1780 - val_acc: 0.9471
Epoch 3/20
60000/60000 [==============================] -
 5s 78us/step - loss: 0.1389 - acc: 0.9593 - v
al_loss: 0.1526 - val_acc: 0.9537
Epoch 4/20
60000/60000 [==============================] -
 5s 85us/step - loss: 0.1136 - acc: 0.9657 - v
al_loss: 0.1324 - val_acc: 0.9591
Epoch 5/20
60000/60000 [==============================] -
 5s 82us/step - loss: 0.0973 - acc: 0.9708 - v
al_loss: 0.1286 - val_acc: 0.9611
Epoch 6/20
60000/60000 [==============================] -
 5s 83us/step - loss: 0.0809 - acc: 0.9749 - v
al_loss: 0.1173 - val_acc: 0.9647
Epoch 7/20
60000/60000 [==============================] -
 5s 85us/step - loss: 0.0675 - acc: 0.9791 - v
al_loss: 0.1133 - val_acc: 0.9642
Epoch 8/20
60000/60000 [==============================] -
 5s 83us/step - loss: 0.0582 - acc: 0.9814 - v
al_loss: 0.1096 - val_acc: 0.9672
```

```
Epoch 9/20
60000/60000 [==============================] -
 5s 86us/step - loss: 0.0516 - acc: 0.9836 - v
al_loss: 0.1088 - val_acc: 0.9674
Epoch 10/20
60000/60000 [==============================] -
 5s 87us/step - loss: 0.0436 - acc: 0.9866 - v
al_loss: 0.1000 - val_acc: 0.9695
Epoch 11/20
60000/60000 [==============================] -
 5s 87us/step - loss: 0.0417 - acc: 0.9863 - v
al_loss: 0.1067 - val_acc: 0.9682
Epoch 12/20
60000/60000 [==============================] -
 5s 87us/step - loss: 0.0321 - acc: 0.9899 - v
al_loss: 0.0923 - val_acc: 0.9734
Epoch 13/20
60000/60000 [==============================] -
 5s 90us/step - loss: 0.0301 - acc: 0.9900 - v
al_loss: 0.0995 - val_acc: 0.9714
Epoch 14/20
60000/60000 [==============================] -
 5s 89us/step - loss: 0.0284 - acc: 0.9907 - v
al_loss: 0.0989 - val_acc: 0.9724
Epoch 15/20
60000/60000 [==============================] -
 5s 88us/step - loss: 0.0262 - acc: 0.9913 - v
al_loss: 0.0993 - val_acc: 0.9716
Epoch 16/20
60000/60000 [==============================] -
 5s 86us/step - loss: 0.0243 - acc: 0.9920 - v
al_loss: 0.1011 - val_acc: 0.9725
Epoch 17/20
60000/60000 [==============================] -
 5s 82us/step - loss: 0.0194 - acc: 0.9938 - v
al_loss: 0.1056 - val_acc: 0.9716
Epoch 18/20
```

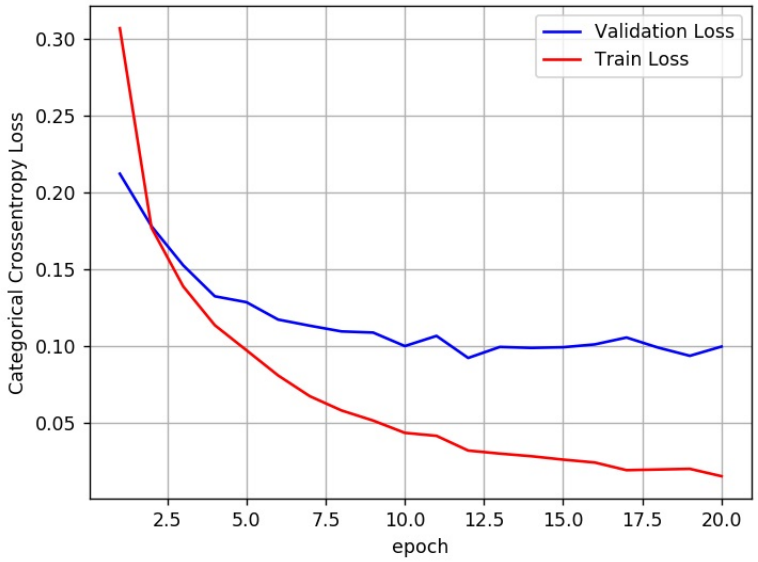
```
60000/60000 [==============================] -
 5s 81us/step - loss: 0.0198 - acc: 0.9933 - v
al_loss: 0.0990 - val_acc: 0.9751
Epoch 19/20
60000/60000 [==============================] -
 5s 82us/step - loss: 0.0202 - acc: 0.9930 - v
al_loss: 0.0937 - val_acc: 0.9756
Epoch 20/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0154 - acc: 0.9951 - v
al_loss: 0.0997 - val_acc: 0.9741
```

In [49]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

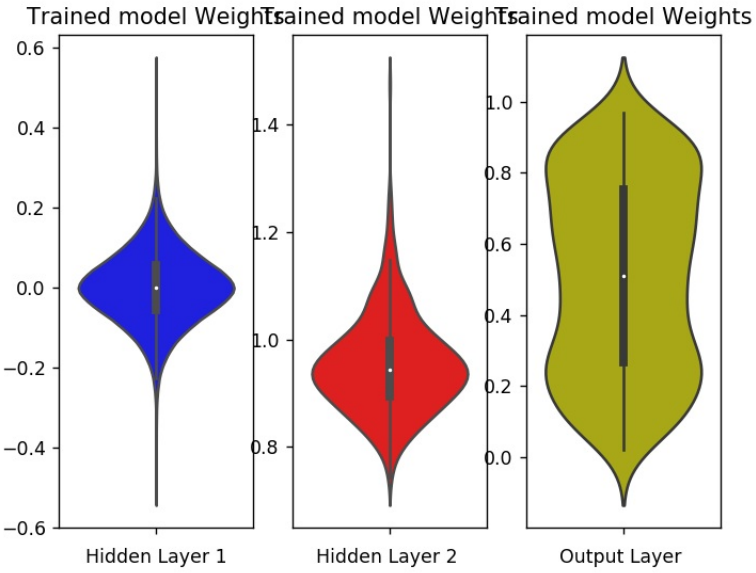
```
Test score: 0.09972703698020195
Test accuracy: 0.9741
```



In [50]:

```
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 [51]:

```
# 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(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(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()
```

```
WARNING:tensorflow:From C:\Users\Public\Anacon
da3\lib\site-packages\keras\backend\tensorflow
_backend.py:3445: calling dropout (from tensor
flow.python.ops.nn_ops) with keep_prob is depr
ecated and will be removed in a future version
.
Instructions for updating:
```

```
Please use `rate` instead of `keep_prob`. Rate
 should be set to `rate = 1 - keep_prob`.
_________________________________________________
_____________________
Layer (type)                 Output Shape
          Param #
=================================================
==================
dense_17 (Dense)             (None, 512)
          401920
_________________________________________________
_____________________
batch_normalization_3 (Batch (None, 512)
          2048
_________________________________________________
_____________________
dropout_1 (Dropout)          (None, 512)
          0
_________________________________________________
_____________________
dense_18 (Dense)             (None, 128)
          65664
_________________________________________________
_____________________
batch_normalization_4 (Batch (None, 128)
          512
_________________________________________________
_____________________
dropout_2 (Dropout)          (None, 128)
          0
_________________________________________________
_____________________
dense_19 (Dense)             (None, 10)
          1290
=================================================
==================
Total params: 471,434
```

```
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________
_____________________
```

In [52]:

```
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 [==============================] - 6s 104us/step - loss: 0.6584 - acc: 0.7964 - val_loss: 0.2836 - val_acc: 0.9142
Epoch 2/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.4215 - acc: 0.8706 - val_loss: 0.2554 - val_acc: 0.9244
Epoch 3/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.3776 - acc: 0.8845 - val_loss: 0.2292 - val_acc: 0.9334
Epoch 4/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.3514 - acc: 0.8932 - val_loss: 0.2255 - val_acc: 0.9324
Epoch 5/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.3320 - acc: 0.8992 - val_loss: 0.2051 - val_acc: 0.9381
Epoch 6/20
60000/60000 [==============================] -
```

```
 6s 92us/step - loss: 0.3184 - acc: 0.9033 - v
al_loss: 0.1978 - val_acc: 0.9413
Epoch 7/20
60000/60000 [==============================] -
 6s 101us/step - loss: 0.3010 - acc: 0.9088 -
val_loss: 0.1902 - val_acc: 0.9445
Epoch 8/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.2923 - acc: 0.9117 - v
al_loss: 0.1782 - val_acc: 0.9487
Epoch 9/20
60000/60000 [==============================] -
 6s 98us/step - loss: 0.2766 - acc: 0.9165 - v
al_loss: 0.1774 - val_acc: 0.9470
Epoch 10/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.2645 - acc: 0.9198 - v
al_loss: 0.1661 - val_acc: 0.9503
Epoch 11/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.2557 - acc: 0.9233 - v
al_loss: 0.1586 - val_acc: 0.9533
Epoch 12/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.2459 - acc: 0.9266 - v
al_loss: 0.1542 - val_acc: 0.9555
Epoch 13/20
60000/60000 [==============================] -
 6s 98us/step - loss: 0.2326 - acc: 0.9286 - v
al_loss: 0.1481 - val_acc: 0.9580
Epoch 14/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.2265 - acc: 0.9311 - v
al_loss: 0.1387 - val_acc: 0.9576
Epoch 15/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.2204 - acc: 0.9329 - v
```

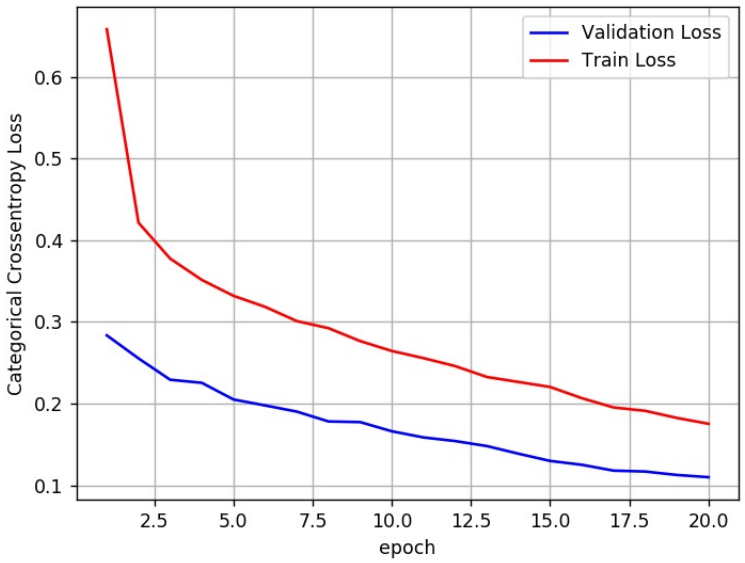
```
al_loss: 0.1299 - val_acc: 0.9595
Epoch 16/20
60000/60000 [==============================] -
 6s 92us/step - loss: 0.2067 - acc: 0.9388 - v
al_loss: 0.1251 - val_acc: 0.9629
Epoch 17/20
60000/60000 [==============================] -
 5s 89us/step - loss: 0.1953 - acc: 0.9405 - v
al_loss: 0.1180 - val_acc: 0.9635
Epoch 18/20
60000/60000 [==============================] -
 5s 92us/step - loss: 0.1912 - acc: 0.9416 - v
al_loss: 0.1169 - val_acc: 0.9648
Epoch 19/20
60000/60000 [==============================] -
 6s 92us/step - loss: 0.1825 - acc: 0.9441 - v
al_loss: 0.1127 - val_acc: 0.9665
Epoch 20/20
60000/60000 [==============================] -
 6s 95us/step - loss: 0.1753 - acc: 0.9470 - v
al_loss: 0.1099 - val_acc: 0.9679
```

In [53]:

```
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 Crossentr
opy Loss')

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

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```

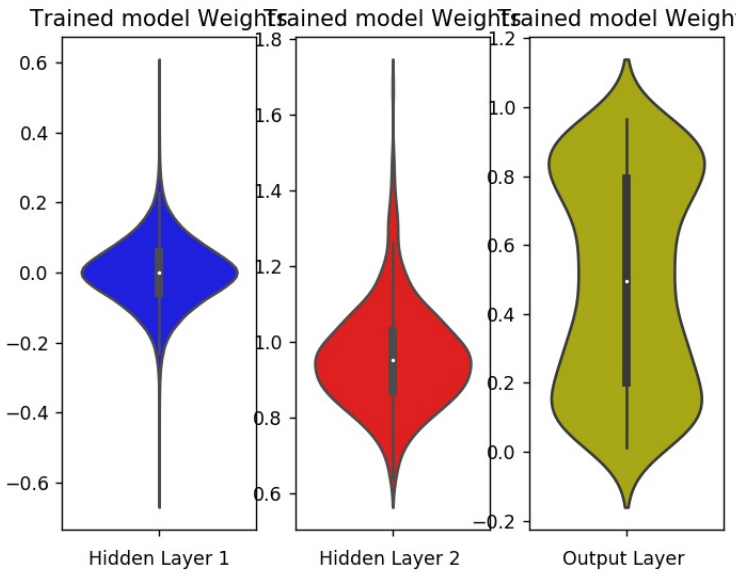
```
# history = model_drop.fit(X_train, Y_train, batch_size=batch
_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

```
Test score: 0.10993219921141863
Test accuracy: 0.9679
```



In [54]:

```
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()
```


Trained model Weights
Trained model Weights
Trained model Weights
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
1.8
1.6
1.4
1.2
1.0
0.8
0.6
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
Hidden Layer 1
Hidden Layer 2
Output Layer

Hyper-parameter tuning of Keras models using Sklearn

In [55]:

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

    model = Sequential()
    model.add(Dense(512, activation=activ, input_shape=(input
_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.06
2, 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'], optimizer='adam')

    return model
```

In [56]:

```
# https://machinelearningmastery.com/grid-search-hyperparamet
ers-deep-learning-models-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_size=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)
```

```
C:\Users\Public\Anaconda3\lib\site-packages\sk
learn\model_selection\_split.py:2053: FutureWa
rning: You should specify a value for 'cv' ins
tead of relying on the default value. The defa
ult value will change from 3 to 5 in version 0
.22.
  warnings.warn(CV_WARNING, FutureWarning)
```

In [57]:

```
print("Best: %f using %s" % (grid_result.best_score_, grid_re
sult.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: 0.974717 using {'activ': 'sigmoid'}
0.974717 (0.001036) with: {'activ': 'sigmoid'}
0.974400 (0.003102) with: {'activ': 'relu'}
```

architecture 2 -hidden layers 3: input (28 *28) hidden layer-1 (512) layer-2 (256) hidden layer-3 (64)output softmax 10

MLP + Sigmoid activation + SGDOptimizer

In [58]:

```
# Multilayer perceptron

model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shap
e=(input_dim,)))
model_sigmoid.add(Dense(256, activation='sigmoid'))
model_sigmoid.add(Dense(64, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()
```

```
_________________________________________________
_________________
Layer (type)                 Output Shape
         Param #
=================================================
==================
dense_41 (Dense)             (None, 512)
         401920
_________________________________________________
_________________
```

```
dense_42 (Dense)             (None, 256)
         131328
_________________________________________________
_________________
dense_43 (Dense)             (None, 64)
         16448
_________________________________________________
_________________
dense_44 (Dense)             (None, 10)
         650
=================================================
==================
Total params: 550,346
Trainable params: 550,346
Non-trainable params: 0
_________________________________________________
_________________
```

In [59]:

```
model_sigmoid.compile(optimizer='sgd', loss='categorical_cros
sentropy', metrics=['accuracy'])

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

```
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 4s 65us/step - loss: 2.3097 - acc: 0.1138 - v
al_loss: 2.2964 - val_acc: 0.1562
Epoch 2/20
60000/60000 [==============================] -
 3s 54us/step - loss: 2.2939 - acc: 0.1208 - v
al_loss: 2.2905 - val_acc: 0.2605
Epoch 3/20
```

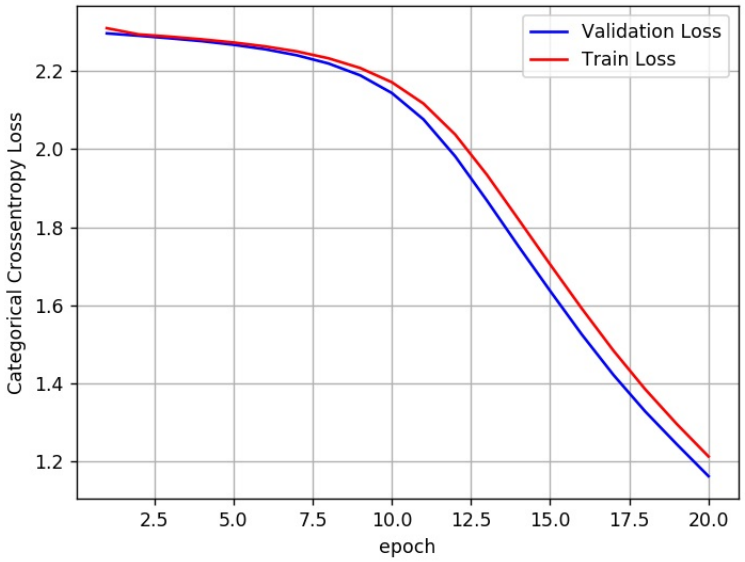
```
60000/60000 [==============================] -
 3s 56us/step - loss: 2.2880 - acc: 0.1296 - v
al_loss: 2.2837 - val_acc: 0.1135
Epoch 4/20
60000/60000 [==============================] -
 4s 61us/step - loss: 2.2812 - acc: 0.1480 - v
al_loss: 2.2765 - val_acc: 0.1489
Epoch 5/20
60000/60000 [==============================] -
 4s 69us/step - loss: 2.2732 - acc: 0.1726 - v
al_loss: 2.2673 - val_acc: 0.2170
Epoch 6/20
60000/60000 [==============================] -
 4s 61us/step - loss: 2.2631 - acc: 0.2102 - v
al_loss: 2.2558 - val_acc: 0.3740
Epoch 7/20
60000/60000 [==============================] -
 4s 60us/step - loss: 2.2502 - acc: 0.2680 - v
al_loss: 2.2403 - val_acc: 0.2942
Epoch 8/20
60000/60000 [==============================] -
 3s 56us/step - loss: 2.2327 - acc: 0.3065 - v
al_loss: 2.2194 - val_acc: 0.3374
Epoch 9/20
60000/60000 [==============================] -
 4s 59us/step - loss: 2.2078 - acc: 0.3329 - v
al_loss: 2.1891 - val_acc: 0.3869
Epoch 10/20
60000/60000 [==============================] -
 4s 65us/step - loss: 2.1714 - acc: 0.3648 - v
al_loss: 2.1438 - val_acc: 0.3916
Epoch 11/20
60000/60000 [==============================] -
 4s 64us/step - loss: 2.1167 - acc: 0.3920 - v
al_loss: 2.0762 - val_acc: 0.3829
Epoch 12/20
60000/60000 [==============================] -
```

```
 4s 59us/step - loss: 2.0376 - acc: 0.4092 - v
al_loss: 1.9818 - val_acc: 0.4213
Epoch 13/20
60000/60000 [==============================] -
 4s 65us/step - loss: 1.9348 - acc: 0.4435 - v
al_loss: 1.8689 - val_acc: 0.4481
Epoch 14/20
60000/60000 [==============================] -
 4s 62us/step - loss: 1.8203 - acc: 0.4743 - v
al_loss: 1.7515 - val_acc: 0.4965
Epoch 15/20
60000/60000 [==============================] -
 3s 57us/step - loss: 1.7049 - acc: 0.5066 - v
al_loss: 1.6371 - val_acc: 0.5206
Epoch 16/20
60000/60000 [==============================] -
 4s 67us/step - loss: 1.5920 - acc: 0.5352 - v
al_loss: 1.5256 - val_acc: 0.5521
Epoch 17/20
60000/60000 [==============================] -
 4s 65us/step - loss: 1.4842 - acc: 0.5632 - v
al_loss: 1.4221 - val_acc: 0.5870
Epoch 18/20
60000/60000 [==============================] -
 4s 59us/step - loss: 1.3856 - acc: 0.5910 - v
al_loss: 1.3288 - val_acc: 0.6094
Epoch 19/20
60000/60000 [==============================] -
 4s 64us/step - loss: 1.2964 - acc: 0.6177 - v
al_loss: 1.2446 - val_acc: 0.6312
Epoch 20/20
60000/60000 [==============================] -
 4s 62us/step - loss: 1.2135 - acc: 0.6421 - v
al_loss: 1.1630 - val_acc: 0.6652
```

In [60]:

```
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_epoch, verbose=1, validation_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)
```

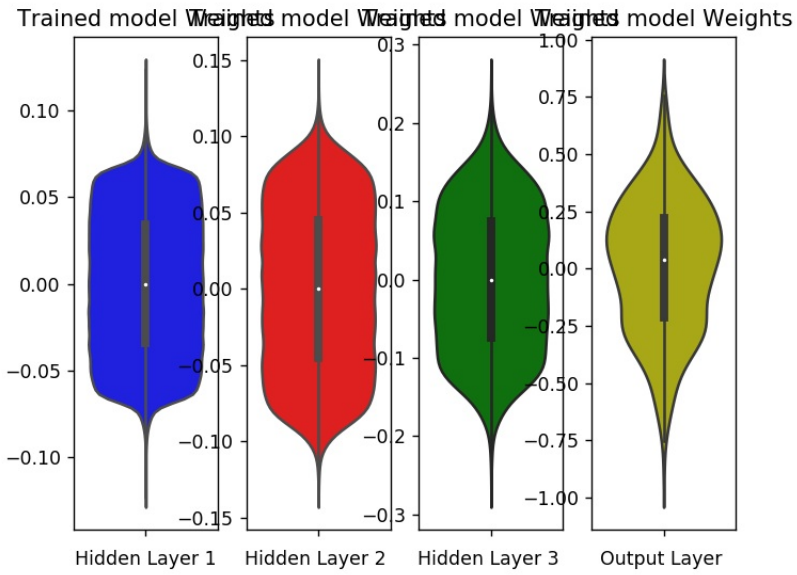
```
Test score: 1.1629849285125733
Test accuracy: 0.6652
```



In [61]:

```
w_after = model_sigmoid.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")

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

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

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

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



MLP + Sigmoid activation + ADAM

In [62]:

```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shap
e=(input_dim,)))
model_sigmoid.add(Dense(256, activation='sigmoid'))
model_sigmoid.add(Dense(64, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()

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

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

```
_________________________________________________________
_________________
Layer (type)                 Output Shape
          Param #
=========================================================
==================
dense_45 (Dense)             (None, 512)
          401920
_________________________________________________________
_________________
dense_46 (Dense)             (None, 256)
          131328
_________________________________________________________
_________________
```

```
dense_47 (Dense)             (None, 64)
         16448
_________________________________________________
_____________________
dense_48 (Dense)             (None, 10)
         650
=================================================
==================
Total params: 550,346
Trainable params: 550,346
Non-trainable params: 0
_________________________________________________
_____________________
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.6544 - acc: 0.8296 - v
al_loss: 0.2533 - val_acc: 0.9288
Epoch 2/20
60000/60000 [==============================] -
 5s 83us/step - loss: 0.2156 - acc: 0.9372 - v
al_loss: 0.1816 - val_acc: 0.9445
Epoch 3/20
60000/60000 [==============================] -
 5s 83us/step - loss: 0.1499 - acc: 0.9562 - v
al_loss: 0.1380 - val_acc: 0.9586
Epoch 4/20
60000/60000 [==============================] -
 5s 86us/step - loss: 0.1127 - acc: 0.9674 - v
al_loss: 0.1036 - val_acc: 0.9675
Epoch 5/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0870 - acc: 0.9746 - v
al_loss: 0.0955 - val_acc: 0.9708
Epoch 6/20
60000/60000 [==============================] -
```

```
 5s 88us/step - loss: 0.0720 - acc: 0.9790 - v
al_loss: 0.0856 - val_acc: 0.9748
Epoch 7/20
60000/60000 [==============================] -
 5s 88us/step - loss: 0.0566 - acc: 0.9832 - v
al_loss: 0.1005 - val_acc: 0.9678
Epoch 8/20
60000/60000 [==============================] -
 5s 86us/step - loss: 0.0452 - acc: 0.9871 - v
al_loss: 0.0793 - val_acc: 0.9754
Epoch 9/20
60000/60000 [==============================] -
 5s 85us/step - loss: 0.0366 - acc: 0.9893 - v
al_loss: 0.0803 - val_acc: 0.9760
Epoch 10/20
60000/60000 [==============================] -
 5s 85us/step - loss: 0.0304 - acc: 0.9910 - v
al_loss: 0.0658 - val_acc: 0.9798
Epoch 11/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0246 - acc: 0.9929 - v
al_loss: 0.0756 - val_acc: 0.9775
Epoch 12/20
60000/60000 [==============================] -
 5s 90us/step - loss: 0.0201 - acc: 0.9945 - v
al_loss: 0.0799 - val_acc: 0.9773
Epoch 13/20
60000/60000 [==============================] -
 5s 89us/step - loss: 0.0159 - acc: 0.9959 - v
al_loss: 0.0721 - val_acc: 0.9803
Epoch 14/20
60000/60000 [==============================] -
 5s 87us/step - loss: 0.0135 - acc: 0.9963 - v
al_loss: 0.0853 - val_acc: 0.9769
Epoch 15/20
60000/60000 [==============================] -
 5s 87us/step - loss: 0.0131 - acc: 0.9962 - v
```

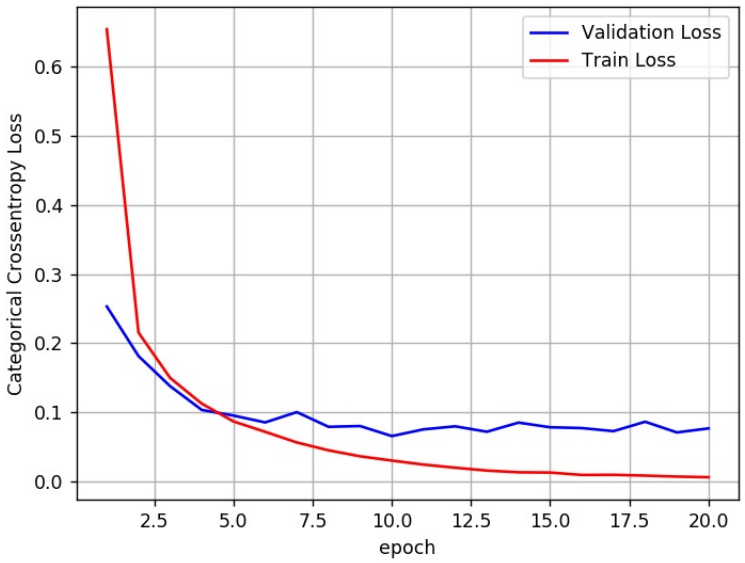
```
al_loss: 0.0786 - val_acc: 0.9783
Epoch 16/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0096 - acc: 0.9976 - v
al_loss: 0.0774 - val_acc: 0.9785
Epoch 17/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0098 - acc: 0.9972 - v
al_loss: 0.0729 - val_acc: 0.9802
Epoch 18/20
60000/60000 [==============================] -
 5s 86us/step - loss: 0.0087 - acc: 0.9975 - v
al_loss: 0.0866 - val_acc: 0.9783
Epoch 19/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0072 - acc: 0.9981 - v
al_loss: 0.0711 - val_acc: 0.9815
Epoch 20/20
60000/60000 [==============================] -
 5s 87us/step - loss: 0.0064 - acc: 0.9983 - v
al_loss: 0.0770 - val_acc: 0.9812
```

In [63]:

```
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 Crossentr
opy Loss')

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

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```

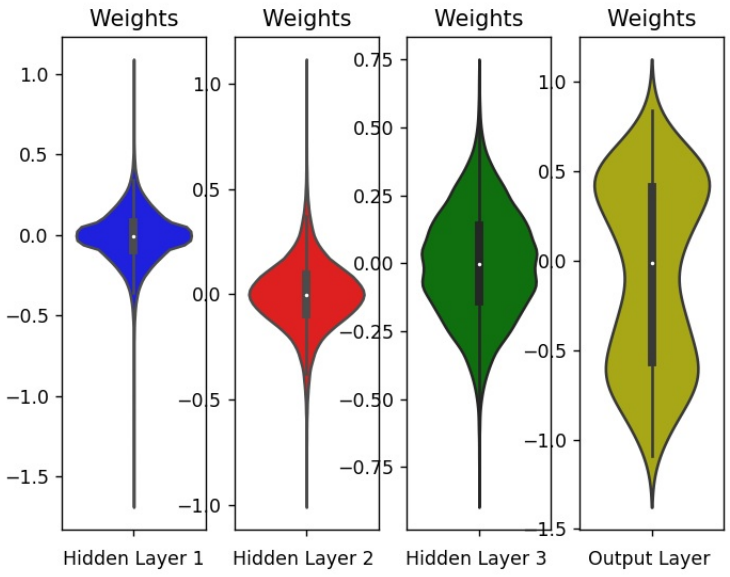
```
# history = model_drop.fit(X_train, Y_train, batch_size=batch
_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

```
Test score: 0.07696367663278943
Test accuracy: 0.9812
```



In [64]:

```
w_after = model_sigmoid.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("Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

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

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


Weights
Weights
Weights
Weights
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5
−1.0
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
1.0
0.5
0.0
−0.5
−1.0
−1.5
Hidden Layer 1
Hidden Layer 2
Hidden Layer 3
Output Layer

MLP + ReLU +SGD

In [65]:

```
# 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 s
atisfy this condition with σ=√(2/(ni).
# h1 =>  σ=√(2/(fan_in) = 0.062  => N(0,σ) = N(0,0.062)
# h2 =>  σ=√(2/(fan_in) = 0.088  => N(0,σ) = N(0,0.088)
# h3 =>  σ=√(2/(fan_in) = 0.176  => N(0,σ) = N(0,0.176)
# out =>  σ=√(2/(fan_in+1) = 0.164  => N(0,σ) = N(0,0.164)

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(inp
ut_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.
062, seed=None)))
model_relu.add(Dense(256, activation='relu', kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.088, seed=None)) )
model_relu.add(Dense(64, activation='relu', kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.176, seed=None)) )

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

model_relu.summary()
```

```
_________________________________________________________
_________________
Layer (type)                 Output Shape
          Param #
=========================================================
==================
```

```
dense_49 (Dense)             (None, 512)
         401920
_________________________________________________
_________________
dense_50 (Dense)             (None, 256)
         131328
_________________________________________________
_________________
dense_51 (Dense)             (None, 64)
         16448
_________________________________________________
_________________
dense_52 (Dense)             (None, 10)
         650
=================================================
==================
Total params: 550,346
Trainable params: 550,346
Non-trainable params: 0
_________________________________________________
_________________
```

In [66]:

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

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 [==============================] - 5s 76us/step - loss: 0.5967 - acc: 0.8265 - val_loss: 0.3096 - val_acc: 0.9089
Epoch 2/20
```

```
60000/60000 [==============================] -
 4s 59us/step - loss: 0.2810 - acc: 0.9181 - v
al_loss: 0.2406 - val_acc: 0.9304
Epoch 3/20
60000/60000 [==============================] -
 4s 65us/step - loss: 0.2270 - acc: 0.9340 - v
al_loss: 0.2068 - val_acc: 0.9388
Epoch 4/20
60000/60000 [==============================] -
 4s 60us/step - loss: 0.1941 - acc: 0.9442 - v
al_loss: 0.1848 - val_acc: 0.9441
Epoch 5/20
60000/60000 [==============================] -
 3s 58us/step - loss: 0.1718 - acc: 0.9505 - v
al_loss: 0.1701 - val_acc: 0.9495
Epoch 6/20
60000/60000 [==============================] -
 4s 59us/step - loss: 0.1550 - acc: 0.9554 - v
al_loss: 0.1567 - val_acc: 0.9522
Epoch 7/20
60000/60000 [==============================] -
 4s 64us/step - loss: 0.1410 - acc: 0.9603 - v
al_loss: 0.1476 - val_acc: 0.9551
Epoch 8/20
60000/60000 [==============================] -
 4s 65us/step - loss: 0.1295 - acc: 0.9632 - v
al_loss: 0.1394 - val_acc: 0.9580
Epoch 9/20
60000/60000 [==============================] -
 4s 64us/step - loss: 0.1197 - acc: 0.9663 - v
al_loss: 0.1330 - val_acc: 0.9590
Epoch 10/20
60000/60000 [==============================] -
 4s 65us/step - loss: 0.1113 - acc: 0.9687 - v
al_loss: 0.1296 - val_acc: 0.9593
Epoch 11/20
60000/60000 [==============================] -
```

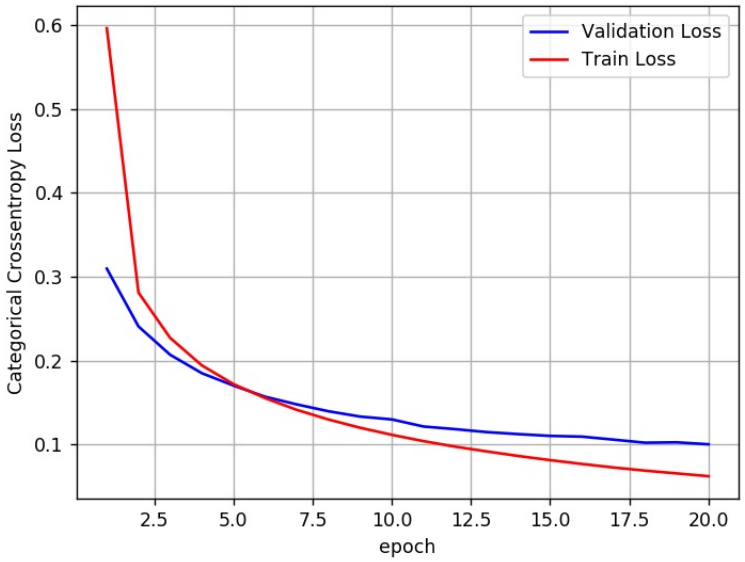
```
 4s 58us/step - loss: 0.1037 - acc: 0.9713 - v
al_loss: 0.1212 - val_acc: 0.9612
Epoch 12/20
60000/60000 [==============================] -
 4s 67us/step - loss: 0.0971 - acc: 0.9729 - v
al_loss: 0.1180 - val_acc: 0.9627
Epoch 13/20
60000/60000 [==============================] -
 4s 64us/step - loss: 0.0914 - acc: 0.9745 - v
al_loss: 0.1145 - val_acc: 0.9637
Epoch 14/20
60000/60000 [==============================] -
 4s 62us/step - loss: 0.0860 - acc: 0.9766 - v
al_loss: 0.1121 - val_acc: 0.9655
Epoch 15/20
60000/60000 [==============================] -
 4s 63us/step - loss: 0.0810 - acc: 0.9776 - v
al_loss: 0.1100 - val_acc: 0.9646
Epoch 16/20
60000/60000 [==============================] -
 4s 62us/step - loss: 0.0765 - acc: 0.9794 - v
al_loss: 0.1091 - val_acc: 0.9649
Epoch 17/20
60000/60000 [==============================] -
 4s 62us/step - loss: 0.0722 - acc: 0.9803 - v
al_loss: 0.1055 - val_acc: 0.9665
Epoch 18/20
60000/60000 [==============================] -
 4s 61us/step - loss: 0.0685 - acc: 0.9815 - v
al_loss: 0.1018 - val_acc: 0.9668
Epoch 19/20
60000/60000 [==============================] -
 4s 61us/step - loss: 0.0652 - acc: 0.9830 - v
al_loss: 0.1023 - val_acc: 0.9676
Epoch 20/20
60000/60000 [==============================] -
 4s 61us/step - loss: 0.0619 - acc: 0.9838 - v
```

```
al_loss: 0.1000 - val_acc: 0.9685
```

In [67]:

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

fig,ax = plt.subplots(1,1)
ax.set_xlabel('epoch') ; ax.set_ylabel('Categorical Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

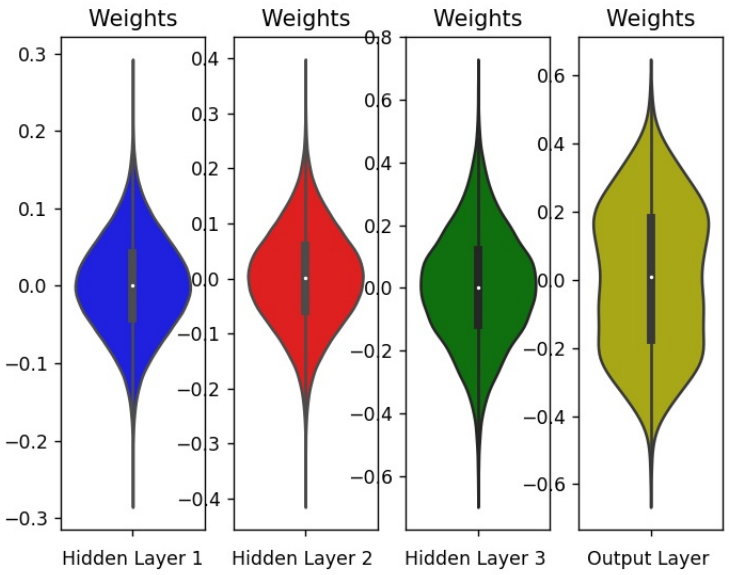
```
Test score: 0.09995875298418104
Test accuracy: 0.9685
```



In [68]:

```
w_after = model_relu.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[2].flatten().reshape(-1,1)
h3_w = w_after[4].flatten().reshape(-1,1)
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("Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

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

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



MLP + ReLU + ADAM

In [69]:

```
model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(inp
ut_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.
062, seed=None)))
model_relu.add(Dense(256, activation='relu', kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.088, seed=None)) )
model_relu.add(Dense(64, activation='relu', kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.176, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_s
ize, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_t
est))
```

```
_________________________________________________________
_________________
Layer (type)                 Output Shape
          Param #
=========================================================
==================
dense_53 (Dense)             (None, 512)
          401920
_________________________________________________________
_________________
dense_54 (Dense)             (None, 256)
```

```
          131328
_________________________________________________
_____________________
dense_55 (Dense)             (None, 64)
          16448
_________________________________________________
_____________________
dense_56 (Dense)             (None, 10)
          650
=================================================
==================
Total params: 550,346
Trainable params: 550,346
Non-trainable params: 0
_________________________________________________
_____________________
None
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.2275 - acc: 0.9318 - v
al_loss: 0.1154 - val_acc: 0.9637
Epoch 2/20
60000/60000 [==============================] -
 5s 89us/step - loss: 0.0838 - acc: 0.9738 - v
al_loss: 0.0851 - val_acc: 0.9739
Epoch 3/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0511 - acc: 0.9840 - v
al_loss: 0.0930 - val_acc: 0.9708
Epoch 4/20
60000/60000 [==============================] -
 5s 86us/step - loss: 0.0390 - acc: 0.9878 - v
al_loss: 0.0712 - val_acc: 0.9768
Epoch 5/20
60000/60000 [==============================] -
```

```
 5s 85us/step - loss: 0.0286 - acc: 0.9902 - v
al_loss: 0.0804 - val_acc: 0.9760
Epoch 6/20
60000/60000 [==============================] -
 5s 88us/step - loss: 0.0233 - acc: 0.9923 - v
al_loss: 0.0780 - val_acc: 0.9787
Epoch 7/20
60000/60000 [==============================] -
 5s 88us/step - loss: 0.0206 - acc: 0.9935 - v
al_loss: 0.0834 - val_acc: 0.9765
Epoch 8/20
60000/60000 [==============================] -
 5s 88us/step - loss: 0.0176 - acc: 0.9941 - v
al_loss: 0.0863 - val_acc: 0.9786
Epoch 9/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0154 - acc: 0.9949 - v
al_loss: 0.0880 - val_acc: 0.9798
Epoch 10/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0182 - acc: 0.9938 - v
al_loss: 0.0997 - val_acc: 0.9769
Epoch 11/20
60000/60000 [==============================] -
 5s 85us/step - loss: 0.0126 - acc: 0.9960 - v
al_loss: 0.0919 - val_acc: 0.9792
Epoch 12/20
60000/60000 [==============================] -
 5s 85us/step - loss: 0.0120 - acc: 0.9958 - v
al_loss: 0.0902 - val_acc: 0.9799
Epoch 13/20
60000/60000 [==============================] -
 5s 83us/step - loss: 0.0159 - acc: 0.9947 - v
al_loss: 0.0885 - val_acc: 0.9781
Epoch 14/20
60000/60000 [==============================] -
 5s 86us/step - loss: 0.0129 - acc: 0.9958 - v
```

```
al_loss: 0.1049 - val_acc: 0.9775
Epoch 15/20
60000/60000 [==============================] -
 5s 83us/step - loss: 0.0058 - acc: 0.9982 - v
al_loss: 0.1280 - val_acc: 0.9757
Epoch 16/20
60000/60000 [==============================] -
 5s 87us/step - loss: 0.0142 - acc: 0.9956 - v
al_loss: 0.1147 - val_acc: 0.9766
Epoch 17/20
60000/60000 [==============================] -
 5s 88us/step - loss: 0.0126 - acc: 0.9961 - v
al_loss: 0.0822 - val_acc: 0.9831
Epoch 18/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0060 - acc: 0.9980 - v
al_loss: 0.0982 - val_acc: 0.9799
Epoch 19/20
60000/60000 [==============================] -
 5s 84us/step - loss: 0.0086 - acc: 0.9971 - v
al_loss: 0.1200 - val_acc: 0.9743
Epoch 20/20
60000/60000 [==============================] -
 5s 85us/step - loss: 0.0140 - acc: 0.9957 - v
al_loss: 0.0907 - val_acc: 0.9807
```

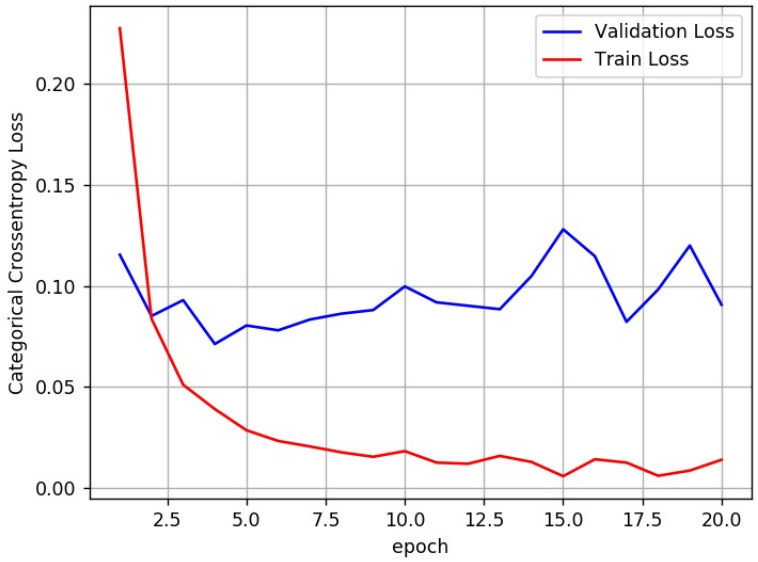
In [70]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs


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

```
Test score: 0.09070437623546104
Test accuracy: 0.9807
```

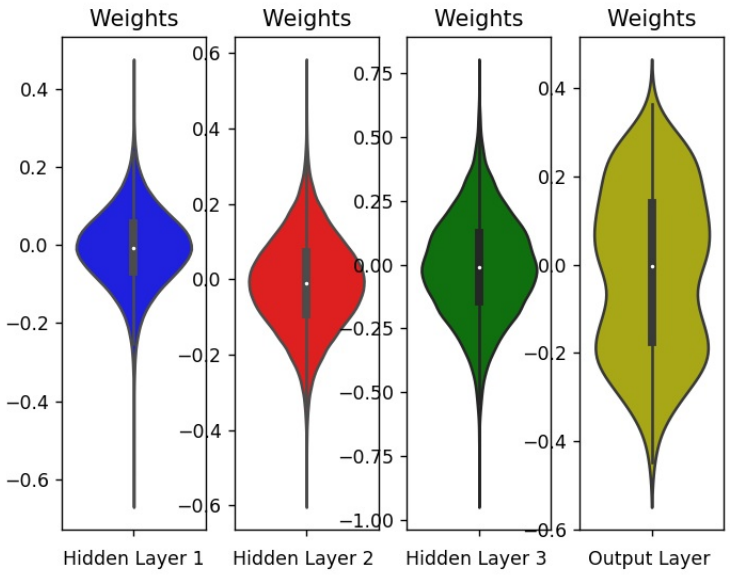
```
C:\Users\Public\Anaconda3\lib\site-packages\ma
tplotlib\pyplot.py:514: RuntimeWarning: More t
han 20 figures have been opened. Figures creat
ed through the pyplot interface (`matplotlib.p
yplot.figure`) are retained until explicitly c
losed and may consume too much memory. (To con
trol this warning, see the rcParam `figure.max
_open_warning`).
  max_open_warning, RuntimeWarning)
```



In [71]:

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

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

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

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



MLP + Batch-Norm on hidden Layers + AdamOptimizer </2>

In [72]:

```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we s
atisfy this condition 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) = 0.051  => N(0,σ) = N(0,0.051)
# h3 =>  σ=√(2/(ni+ni+1) = 0.079  => N(0,σ) = N(0,0.079)

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, stdde
v=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(256, activation='sigmoid', kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.051, seed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(64, activation='sigmoid', kernel_initia
lizer=RandomNormal(mean=0.0, stddev=0.079, seed=None)) )
model_batch.add(BatchNormalization())

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

```
model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_57 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_5 (Batch (None, 512)               2048
_________________________________________________________________
dense_58 (Dense)             (None, 256)               131328
_________________________________________________________________
batch_normalization_6 (Batch (None, 256)               1024
_________________________________________________________________
dense_59 (Dense)             (None, 64)                16448
_________________________________________________________________
batch_normalization_7 (Batch (None, 64)                256
_________________________________________________________________
dense_60 (Dense)             (None, 10)                650
=================================================================
```

```
Total params: 553,674
Trainable params: 552,010
Non-trainable params: 1,664
_________________________________________________________
_____________________
```

In [73]:

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

history = model_batch.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 samp
les
Epoch 1/20
60000/60000 [==============================] -
 8s 130us/step - loss: 0.2636 - acc: 0.9242 -
val_loss: 0.1515 - val_acc: 0.9575
Epoch 2/20
60000/60000 [==============================] -
 6s 103us/step - loss: 0.1288 - acc: 0.9616 -
val_loss: 0.1129 - val_acc: 0.9672
Epoch 3/20
60000/60000 [==============================] -
 6s 103us/step - loss: 0.0869 - acc: 0.9739 -
val_loss: 0.0858 - val_acc: 0.9748
Epoch 4/20
60000/60000 [==============================] -
 6s 100us/step - loss: 0.0623 - acc: 0.9806 -
val_loss: 0.0942 - val_acc: 0.9704
Epoch 5/20
60000/60000 [==============================] -
 6s 93us/step - loss: 0.0477 - acc: 0.9850 - v
al_loss: 0.0814 - val_acc: 0.9749
Epoch 6/20
```

```
60000/60000 [==============================] -
 6s 100us/step - loss: 0.0365 - acc: 0.9885 -
val_loss: 0.0719 - val_acc: 0.9779
Epoch 7/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0361 - acc: 0.9884 - v
al_loss: 0.0780 - val_acc: 0.9752
Epoch 8/20
60000/60000 [==============================] -
 6s 100us/step - loss: 0.0263 - acc: 0.9913 -
val_loss: 0.0771 - val_acc: 0.9776
Epoch 9/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.0225 - acc: 0.9923 - v
al_loss: 0.0805 - val_acc: 0.9780
Epoch 10/20
60000/60000 [==============================] -
 6s 106us/step - loss: 0.0217 - acc: 0.9927 -
val_loss: 0.0893 - val_acc: 0.9748
Epoch 11/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0199 - acc: 0.9931 - v
al_loss: 0.0812 - val_acc: 0.9770
Epoch 12/20
60000/60000 [==============================] -
 6s 100us/step - loss: 0.0161 - acc: 0.9945 -
val_loss: 0.0981 - val_acc: 0.9731
Epoch 13/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0155 - acc: 0.9947 - v
al_loss: 0.0855 - val_acc: 0.9760
Epoch 14/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0157 - acc: 0.9948 - v
al_loss: 0.0783 - val_acc: 0.9782
Epoch 15/20
60000/60000 [==============================] -
```

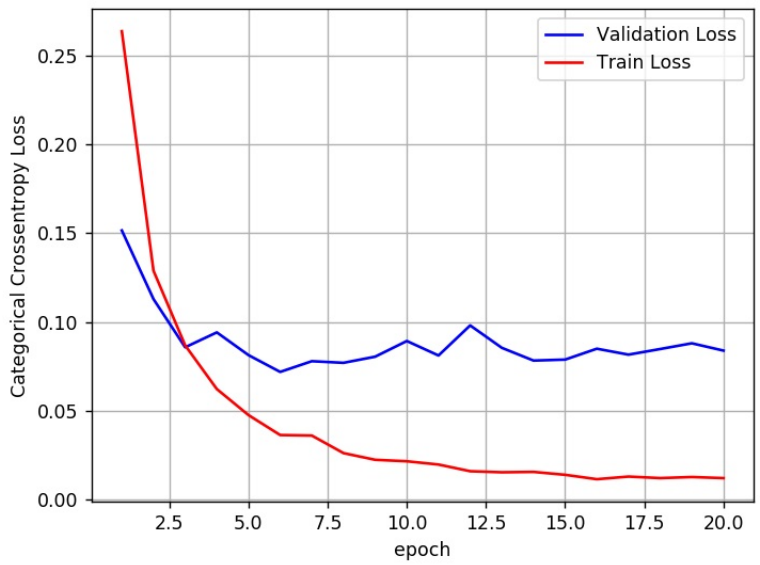
```
 6s 100us/step - loss: 0.0141 - acc: 0.9952 -
val_loss: 0.0789 - val_acc: 0.9797
Epoch 16/20
60000/60000 [==============================] -
 6s 98us/step - loss: 0.0116 - acc: 0.9961 - v
al_loss: 0.0850 - val_acc: 0.9781
Epoch 17/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0131 - acc: 0.9957 - v
al_loss: 0.0817 - val_acc: 0.9802
Epoch 18/20
60000/60000 [==============================] -
 6s 107us/step - loss: 0.0122 - acc: 0.9957 -
val_loss: 0.0848 - val_acc: 0.9790
Epoch 19/20
60000/60000 [==============================] -
 6s 101us/step - loss: 0.0128 - acc: 0.9955 -
val_loss: 0.0880 - val_acc: 0.9794
Epoch 20/20
60000/60000 [==============================] -
 6s 107us/step - loss: 0.0122 - acc: 0.9957 -
val_loss: 0.0839 - val_acc: 0.9790
```

In [74]:

```
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 Crossentr
opy Loss')

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

# print(history.history.keys())
```

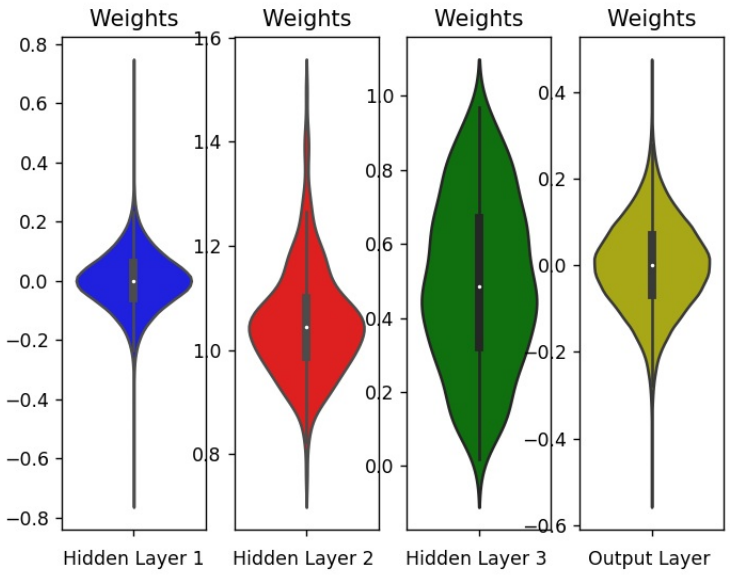
```
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch
_size, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

```
Test score: 0.08393007170943602
Test accuracy: 0.979
```



In [75]:

```
w_after = model_batch.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")

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

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

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

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


Weights
Weights
Weights
Weights
Hidden Layer 1
Hidden Layer 2
Hidden Layer 3
Output Layer

5. MLP + Dropout + AdamOptimizer

In [76]:

```
# 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(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(256, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.0, stddev=0.051, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(64, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.0, stddev=0.079, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

_____

_____

```
Layer (type)                 Output Shape
         Param #
=================================================
==================
dense_61 (Dense)             (None, 512)
         401920
_________________________________________________
_____________________
batch_normalization_8 (Batch (None, 512)
         2048
_________________________________________________
_____________________
dropout_3 (Dropout)          (None, 512)
         0
_________________________________________________
_____________________
dense_62 (Dense)             (None, 256)
         131328
_________________________________________________
_____________________
batch_normalization_9 (Batch (None, 256)
         1024
_________________________________________________
_____________________
dropout_4 (Dropout)          (None, 256)
         0
_________________________________________________
_____________________
dense_63 (Dense)             (None, 64)
         16448
_________________________________________________
_____________________
batch_normalization_10 (Batc (None, 64)
         256
_________________________________________________
_____________________
dropout_5 (Dropout)          (None, 64)
```

```
         0
_________________________________________________________________
_________________
dense_64 (Dense)             (None, 10)
         650
=================================================================
==================
Total params: 553,674
Trainable params: 552,010
Non-trainable params: 1,664
_________________________________________________________________
_________________
```

In [77]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_s
ize, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_t
est))
```

```
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 9s 144us/step - loss: 0.5961 - acc: 0.8192 -
val_loss: 0.2633 - val_acc: 0.9214
Epoch 2/20
60000/60000 [==============================] -
 7s 116us/step - loss: 0.3927 - acc: 0.8853 -
val_loss: 0.2321 - val_acc: 0.9319
Epoch 3/20
60000/60000 [==============================] -
 7s 119us/step - loss: 0.3416 - acc: 0.9001 -
val_loss: 0.2031 - val_acc: 0.9399
Epoch 4/20
60000/60000 [==============================] -
```

```
 7s 114us/step - loss: 0.3048 - acc: 0.9110 -
val_loss: 0.1796 - val_acc: 0.9491
Epoch 5/20
60000/60000 [==============================] -
 7s 114us/step - loss: 0.2755 - acc: 0.9202 -
val_loss: 0.1636 - val_acc: 0.9520
Epoch 6/20
60000/60000 [==============================] -
 7s 118us/step - loss: 0.2457 - acc: 0.9291 -
val_loss: 0.1442 - val_acc: 0.9571
Epoch 7/20
60000/60000 [==============================] -
 7s 117us/step - loss: 0.2292 - acc: 0.9338 -
val_loss: 0.1306 - val_acc: 0.9610
Epoch 8/20
60000/60000 [==============================] -
 7s 113us/step - loss: 0.2069 - acc: 0.9400 -
val_loss: 0.1226 - val_acc: 0.9638
Epoch 9/20
60000/60000 [==============================] -
 7s 113us/step - loss: 0.1890 - acc: 0.9451 -
val_loss: 0.1113 - val_acc: 0.9681
Epoch 10/20
60000/60000 [==============================] -
 7s 110us/step - loss: 0.1775 - acc: 0.9490 -
val_loss: 0.1005 - val_acc: 0.9714
Epoch 11/20
60000/60000 [==============================] -
 7s 110us/step - loss: 0.1631 - acc: 0.9534 -
val_loss: 0.0971 - val_acc: 0.9720
Epoch 12/20
60000/60000 [==============================] -
 7s 116us/step - loss: 0.1509 - acc: 0.9572 -
val_loss: 0.0903 - val_acc: 0.9743
Epoch 13/20
60000/60000 [==============================] -
 7s 122us/step - loss: 0.1459 - acc: 0.9579 -
```

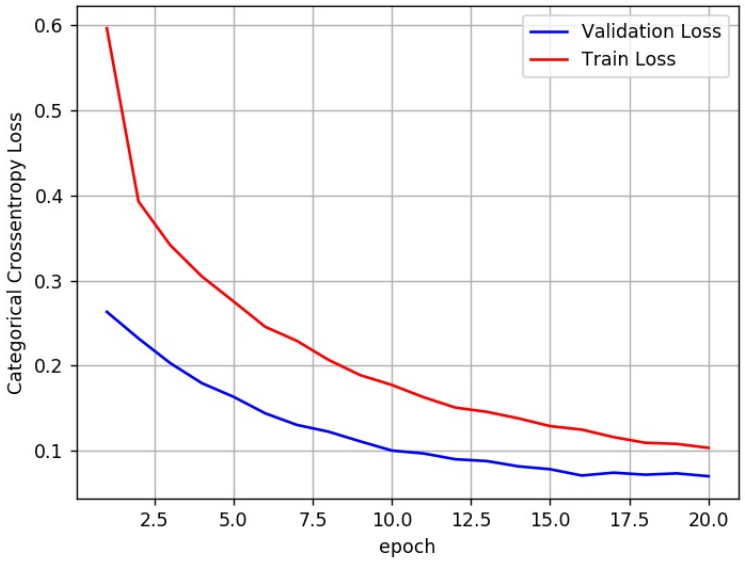
```
val_loss: 0.0881 - val_acc: 0.9754
Epoch 14/20
60000/60000 [==============================] -
 7s 113us/step - loss: 0.1382 - acc: 0.9595 -
val_loss: 0.0819 - val_acc: 0.9762
Epoch 15/20
60000/60000 [==============================] -
 7s 111us/step - loss: 0.1291 - acc: 0.9618 -
val_loss: 0.0786 - val_acc: 0.9776
Epoch 16/20
60000/60000 [==============================] -
 7s 114us/step - loss: 0.1250 - acc: 0.9643 -
val_loss: 0.0712 - val_acc: 0.9796
Epoch 17/20
60000/60000 [==============================] -
 7s 113us/step - loss: 0.1163 - acc: 0.9667 -
val_loss: 0.0745 - val_acc: 0.9791
Epoch 18/20
60000/60000 [==============================] -
 7s 111us/step - loss: 0.1096 - acc: 0.9688 -
val_loss: 0.0721 - val_acc: 0.9801
Epoch 19/20
60000/60000 [==============================] -
 7s 114us/step - loss: 0.1083 - acc: 0.9690 -
val_loss: 0.0737 - val_acc: 0.9798
Epoch 20/20
60000/60000 [==============================] -
 7s 112us/step - loss: 0.1036 - acc: 0.9700 -
val_loss: 0.0704 - val_acc: 0.9802
```

In [78]:

```
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, validation_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)
```

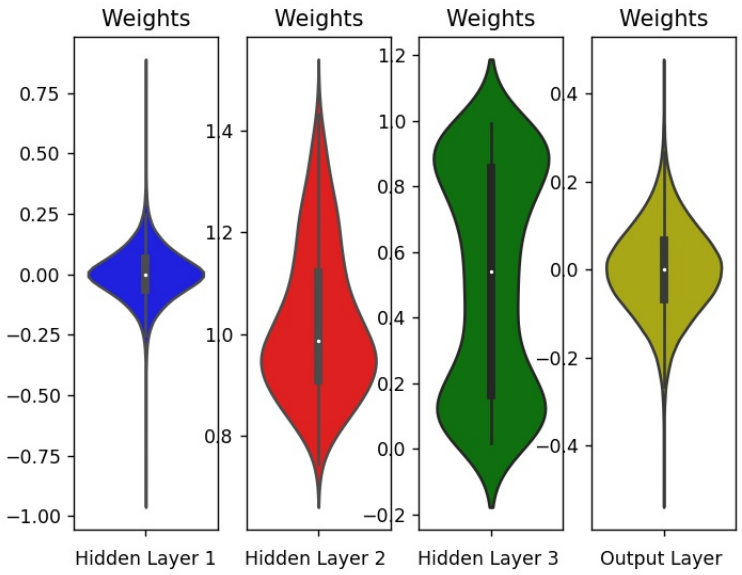
```
Test score: 0.0703752794698812
Test accuracy: 0.9802
```



In [79]:

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

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

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

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



Hyper-parameter tuning of Keras models using Sklearn

In [80]:

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

    model = Sequential()
    model.add(Dense(512, activation=activ, input_shape=(input
_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.06
2, seed=None)))
    model.add(Dense(256, activation=activ, kernel_initializer
=RandomNormal(mean=0.0, stddev=0.088, seed=None)) )
    model.add(Dense(64, activation=activ, kernel_initializer=
RandomNormal(mean=0.0, stddev=0.176, seed=None)) )
    model.add(Dense(output_dim, activation='softmax'))


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

    return model
```

In [81]:

```
# https://machinelearningmastery.com/grid-search-hyperparamet
ers-deep-learning-models-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_size=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 [82]:

```
print("Best: %f using %s" % (grid_result.best_score_, grid_re
sult.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: 0.975400 using {'activ': 'sigmoid'}
0.975400 (0.001167) with: {'activ': 'sigmoid'}
0.974300 (0.002046) with: {'activ': 'relu'}
```

architecture 3 -hidden layers 5: input (28 *28) hidden layer-1 (512) layer-2 (256) hidden layer-3 (128) hidden layer-4 (64) hidden layer-5 (32) output softmax 10

MLP + Sigmoid activation + SGDOptimizer

In [83]:

```
# Multilayer perceptron

model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shap
e=(input_dim,)))
model_sigmoid.add(Dense(256, activation='sigmoid'))
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(64, activation='sigmoid'))
model_sigmoid.add(Dense(32, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()
```

```
_________________________________________________
_________________
Layer (type)                 Output Shape
          Param #
=================================================
==================
dense_93 (Dense)             (None, 512)
```

```
          401920
_________________________________________________________________
dense_94 (Dense)             (None, 256)
          131328
_________________________________________________________________
dense_95 (Dense)             (None, 128)
          32896
_________________________________________________________________
dense_96 (Dense)             (None, 64)
          8256
_________________________________________________________________
dense_97 (Dense)             (None, 32)
          2080
_________________________________________________________________
dense_98 (Dense)             (None, 10)
          330
=================================================================
Total params: 576,810
Trainable params: 576,810
Non-trainable params: 0
_________________________________________________________________
```

In [84]:

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

history = model_sigmoid.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 samp
les
Epoch 1/20
60000/60000 [==============================] -
 5s 91us/step - loss: 2.3155 - acc: 0.1124 - v
al_loss: 2.3012 - val_acc: 0.1135
Epoch 2/20
60000/60000 [==============================] -
 4s 66us/step - loss: 2.3014 - acc: 0.1124 - v
al_loss: 2.3012 - val_acc: 0.1135
Epoch 3/20
60000/60000 [==============================] -
 4s 68us/step - loss: 2.3014 - acc: 0.1124 - v
al_loss: 2.3009 - val_acc: 0.1135
Epoch 4/20
60000/60000 [==============================] -
 4s 66us/step - loss: 2.3013 - acc: 0.1124 - v
al_loss: 2.3009 - val_acc: 0.1135
Epoch 5/20
60000/60000 [==============================] -
 4s 66us/step - loss: 2.3013 - acc: 0.1124 - v
al_loss: 2.3011 - val_acc: 0.1135
Epoch 6/20
60000/60000 [==============================] -
 4s 67us/step - loss: 2.3013 - acc: 0.1124 - v
al_loss: 2.3010 - val_acc: 0.1135
Epoch 7/20
60000/60000 [==============================] -
 4s 71us/step - loss: 2.3012 - acc: 0.1124 - v
al_loss: 2.3008 - val_acc: 0.1135
Epoch 8/20
60000/60000 [==============================] -
 4s 69us/step - loss: 2.3012 - acc: 0.1124 - v
al_loss: 2.3011 - val_acc: 0.1135
Epoch 9/20
60000/60000 [==============================] -
 4s 68us/step - loss: 2.3012 - acc: 0.1124 - v
```

```
al_loss: 2.3010 - val_acc: 0.1135
Epoch 10/20
60000/60000 [==============================] -
 4s 68us/step - loss: 2.3012 - acc: 0.1123 - v
al_loss: 2.3009 - val_acc: 0.1135
Epoch 11/20
60000/60000 [==============================] -
 4s 73us/step - loss: 2.3011 - acc: 0.1124 - v
al_loss: 2.3011 - val_acc: 0.1135
Epoch 12/20
60000/60000 [==============================] -
 4s 71us/step - loss: 2.3011 - acc: 0.1124 - v
al_loss: 2.3007 - val_acc: 0.1135
Epoch 13/20
60000/60000 [==============================] -
 4s 71us/step - loss: 2.3010 - acc: 0.1124 - v
al_loss: 2.3010 - val_acc: 0.1135
Epoch 14/20
60000/60000 [==============================] -
 4s 72us/step - loss: 2.3011 - acc: 0.1124 - v
al_loss: 2.3009 - val_acc: 0.1135
Epoch 15/20
60000/60000 [==============================] -
 4s 73us/step - loss: 2.3010 - acc: 0.1124 - v
al_loss: 2.3009 - val_acc: 0.1135
Epoch 16/20
60000/60000 [==============================] -
 4s 71us/step - loss: 2.3010 - acc: 0.1124 - v
al_loss: 2.3008 - val_acc: 0.1135
Epoch 17/20
60000/60000 [==============================] -
 4s 71us/step - loss: 2.3010 - acc: 0.1124 - v
al_loss: 2.3008 - val_acc: 0.1135
Epoch 18/20
60000/60000 [==============================] -
 4s 72us/step - loss: 2.3009 - acc: 0.1124 - v
al_loss: 2.3006 - val_acc: 0.1135
```

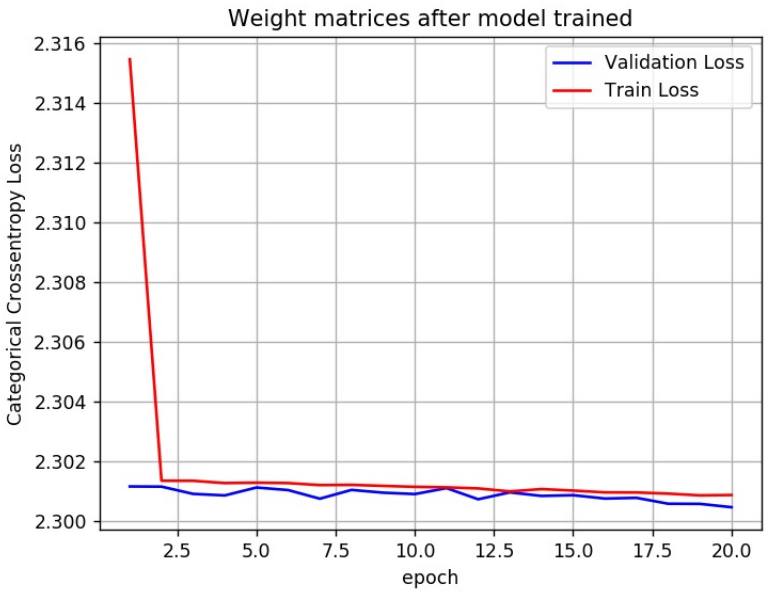
```
Epoch 19/20
60000/60000 [==============================] -
 4s 72us/step - loss: 2.3009 - acc: 0.1124 - v
al_loss: 2.3006 - val_acc: 0.1135
Epoch 20/20
60000/60000 [==============================] -
 4s 71us/step - loss: 2.3009 - acc: 0.1124 - v
al_loss: 2.3005 - val_acc: 0.1135
```

In [85]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs
```

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

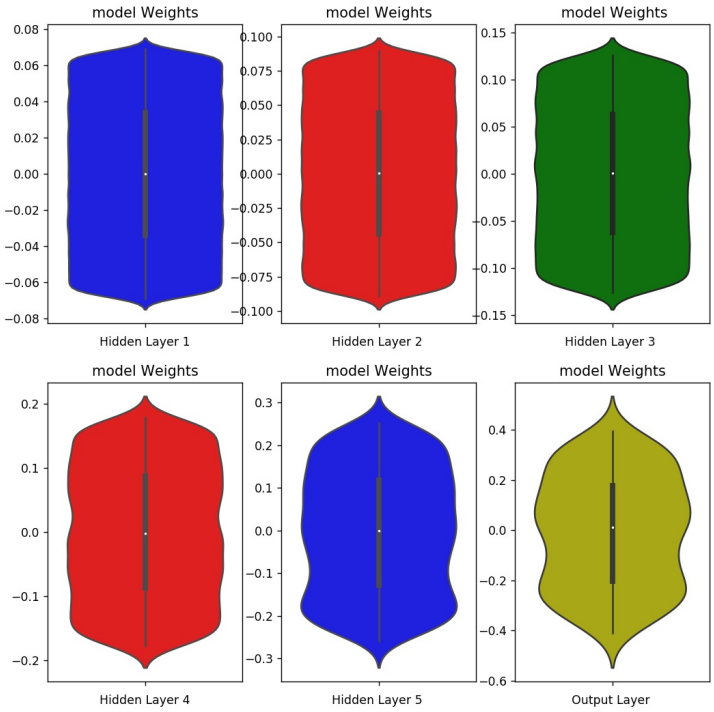
```
Test score: 2.3004731758117676
Test accuracy: 0.1135
```



In [93]:

```
w_after = model_sigmoid.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)

plt.title("Weight matrices after model trained")
fig = plt.figure(figsize=(10,10))
```

```
plt.subplot(2, 3, 1)
plt.title("model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

plt.subplot(2, 3, 3)
plt.title(" model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(2, 3, 4)
plt.title("model Weights")
ax = sns.violinplot(y=h4_w,color='r')
plt.xlabel('Hidden Layer 4')

plt.subplot(2, 3, 5)
plt.title("model Weights")
ax = sns.violinplot(y=h5_w, color='b')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(2, 3, 6)
plt.title("model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```


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

MLP + Sigmoid activation + ADAM

In [94]:

```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shap
e=(input_dim,)))
model_sigmoid.add(Dense(256, activation='sigmoid'))
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(64, activation='sigmoid'))
model_sigmoid.add(Dense(32, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()

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

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

```
_________________________________________________________
_____________________
Layer (type)                 Output Shape
          Param #
=========================================================
==================
dense_99 (Dense)             (None, 512)
          401920
_________________________________________________________
_____________________
dense_100 (Dense)            (None, 256)
          131328
```

```
_________________________________________________
_____________________
dense_101 (Dense)            (None, 128)
         32896
_________________________________________________
_____________________
dense_102 (Dense)            (None, 64)
         8256
_________________________________________________
_____________________
dense_103 (Dense)            (None, 32)
         2080
_________________________________________________
_____________________
dense_104 (Dense)            (None, 10)
         330
=================================================
==================
Total params: 576,810
Trainable params: 576,810
Non-trainable params: 0
_________________________________________________
_____________________
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 7s 122us/step - loss: 1.2380 - acc: 0.6232 -
val_loss: 0.5369 - val_acc: 0.8743
Epoch 2/20
60000/60000 [==============================] -
 6s 95us/step - loss: 0.3561 - acc: 0.9166 - v
al_loss: 0.2575 - val_acc: 0.9386
Epoch 3/20
60000/60000 [==============================] -
 6s 92us/step - loss: 0.2012 - acc: 0.9512 - v
al_loss: 0.1934 - val_acc: 0.9513
```

```
Epoch 4/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.1457 - acc: 0.9628 - v
al_loss: 0.1535 - val_acc: 0.9605
Epoch 5/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.1150 - acc: 0.9700 - v
al_loss: 0.1309 - val_acc: 0.9659
Epoch 6/20
60000/60000 [==============================] -
 6s 92us/step - loss: 0.0915 - acc: 0.9762 - v
al_loss: 0.1161 - val_acc: 0.9681
Epoch 7/20
60000/60000 [==============================] -
 6s 95us/step - loss: 0.0762 - acc: 0.9798 - v
al_loss: 0.1068 - val_acc: 0.9704
Epoch 8/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0638 - acc: 0.9833 - v
al_loss: 0.1004 - val_acc: 0.9716
Epoch 9/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.0531 - acc: 0.9860 - v
al_loss: 0.0984 - val_acc: 0.9748
Epoch 10/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.0477 - acc: 0.9871 - v
al_loss: 0.0944 - val_acc: 0.9744
Epoch 11/20
60000/60000 [==============================] -
 6s 101us/step - loss: 0.0396 - acc: 0.9896 -
val_loss: 0.0867 - val_acc: 0.9776
Epoch 12/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.0348 - acc: 0.9905 - v
al_loss: 0.0888 - val_acc: 0.9755
Epoch 13/20
```

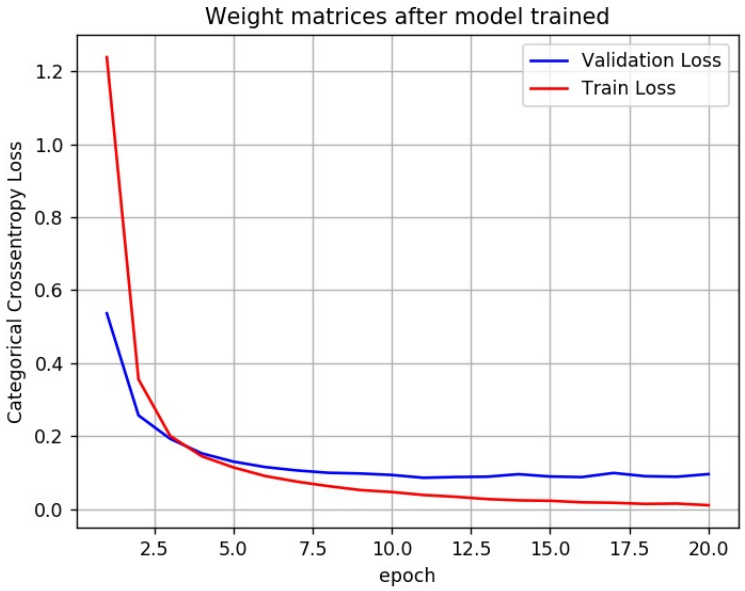
```
60000/60000 [==============================] -
 6s 98us/step - loss: 0.0285 - acc: 0.9925 - v
al_loss: 0.0896 - val_acc: 0.9771
Epoch 14/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.0250 - acc: 0.9931 - v
al_loss: 0.0965 - val_acc: 0.9744
Epoch 15/20
60000/60000 [==============================] -
 6s 94us/step - loss: 0.0238 - acc: 0.9933 - v
al_loss: 0.0902 - val_acc: 0.9771
Epoch 16/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.0195 - acc: 0.9947 - v
al_loss: 0.0886 - val_acc: 0.9771
Epoch 17/20
60000/60000 [==============================] -
 6s 93us/step - loss: 0.0183 - acc: 0.9952 - v
al_loss: 0.0999 - val_acc: 0.9750
Epoch 18/20
60000/60000 [==============================] -
 6s 94us/step - loss: 0.0152 - acc: 0.9958 - v
al_loss: 0.0910 - val_acc: 0.9790
Epoch 19/20
60000/60000 [==============================] -
 6s 100us/step - loss: 0.0161 - acc: 0.9957 -
val_loss: 0.0896 - val_acc: 0.9798
Epoch 20/20
60000/60000 [==============================] -
 6s 98us/step - loss: 0.0116 - acc: 0.9967 - v
al_loss: 0.0967 - val_acc: 0.9781
```

In [95]:

```
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_epoch, verbose=1, validation_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)
```

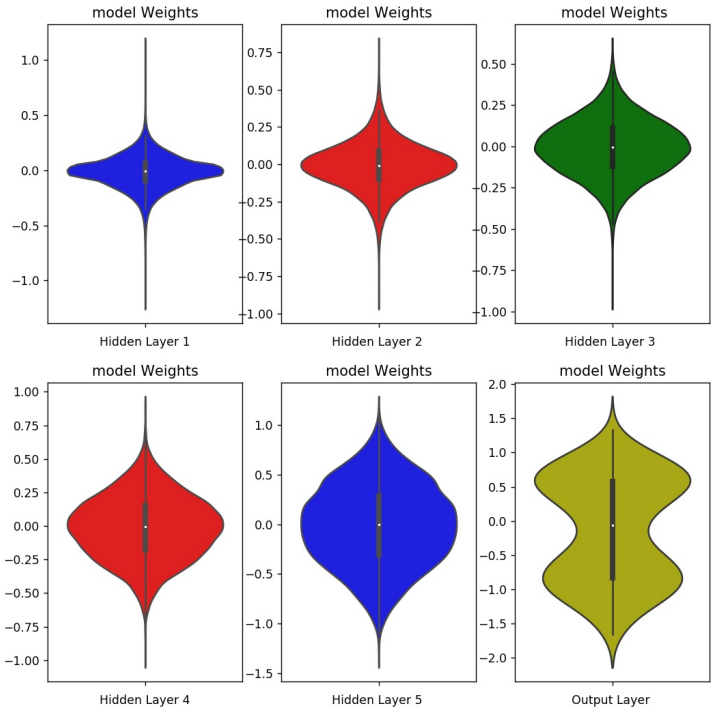
```
Test score: 0.09669930382110178
Test accuracy: 0.9781
```



In [96]:

```
w_after = model_sigmoid.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)

plt.title("Weight matrices after model trained")
fig = plt.figure(figsize=(10,10))


plt.subplot(2, 3, 1)
plt.title("model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(2, 3, 2)
plt.title("model Weights")
```

```
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(2, 3, 3)
plt.title(" model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(2, 3, 4)
plt.title("model Weights")
ax = sns.violinplot(y=h4_w,color='r')
plt.xlabel('Hidden Layer 4')

plt.subplot(2, 3, 5)
plt.title("model Weights")
ax = sns.violinplot(y=h5_w, color='b')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(2, 3, 6)
plt.title("model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```


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

MLP + ReLU +SGD

In [97]:

```
# 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 s
atisfy this condition with σ=√(2/(ni).
# h1 =>  σ=√(2/(fan_in) = 0.062  => N(0,σ) = N(0,0.062)
# h2 =>  σ=√(2/(fan_in) = 0.088  => N(0,σ) = N(0,0.088)
# h3 =>  σ=√(2/(fan_in) = 0.125  => N(0,σ) = N(0,0.125)
#h4 =>  σ=√(2/(fan_in) = 0.176 => N(0,σ) = N(0,0.176)
# h5 =>  σ=√(2/(fan_in) = 0.25  => N(0,σ) = N(0,0.25)

# out =>  σ=√(2/(fan_in+1) = 0.218  => N(0,σ) = N(0,0.218)

model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(inp
ut_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.
062, seed=None)))
model_relu.add(Dense(256, activation='relu', kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.088, seed=None)) )
model_relu.add(Dense(128, activation='relu', kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.125, seed=None)) )
model_relu.add(Dense(64, activation='relu', kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.176, seed=None)) )
model_relu.add(Dense(32, activation='relu', kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.25, seed=None)) )

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

model_relu.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_105 (Dense)            (None, 512)               401920
_________________________________________________________________
dense_106 (Dense)            (None, 256)               131328
_________________________________________________________________
dense_107 (Dense)            (None, 128)               32896
_________________________________________________________________
dense_108 (Dense)            (None, 64)                8256
_________________________________________________________________
dense_109 (Dense)            (None, 32)                2080
_________________________________________________________________
dense_110 (Dense)            (None, 10)                330
=================================================================
Total params: 576,810
Trainable params: 576,810
Non-trainable params: 0
_________________________________________________________________
```

In [98]:

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

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 [==============================] - 6s 94us/step - loss: 0.5969 - acc: 0.8209 - val_loss: 0.2808 - val_acc: 0.9167
Epoch 2/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.2519 - acc: 0.9246 - val_loss: 0.2160 - val_acc: 0.9363
Epoch 3/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.1956 - acc: 0.9420 - val_loss: 0.1938 - val_acc: 0.9420
Epoch 4/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.1647 - acc: 0.9514 - val_loss: 0.1948 - val_acc: 0.9415
Epoch 5/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.1431 - acc: 0.9576 - val_loss: 0.1582 - val_acc: 0.9490
Epoch 6/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1254 - acc: 0.9627 - val_loss: 0.1460 - val_acc: 0.9542
Epoch 7/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1126 - acc: 0.9663 - val_loss: 0.1500 - val_acc: 0.9543
```

```
Epoch 8/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.1018 - acc: 0.9697 - v
al_loss: 0.1372 - val_acc: 0.9592
Epoch 9/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0905 - acc: 0.9738 - v
al_loss: 0.1278 - val_acc: 0.9618
Epoch 10/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0829 - acc: 0.9758 - v
al_loss: 0.1254 - val_acc: 0.9629
Epoch 11/20
60000/60000 [==============================] -
 4s 69us/step - loss: 0.0753 - acc: 0.9781 - v
al_loss: 0.1187 - val_acc: 0.9640
Epoch 12/20
60000/60000 [==============================] -
 4s 73us/step - loss: 0.0690 - acc: 0.9803 - v
al_loss: 0.1258 - val_acc: 0.9618
Epoch 13/20
60000/60000 [==============================] -
 4s 71us/step - loss: 0.0631 - acc: 0.9821 - v
al_loss: 0.1145 - val_acc: 0.9659
Epoch 14/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0579 - acc: 0.9840 - v
al_loss: 0.1198 - val_acc: 0.9647
Epoch 15/20
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0526 - acc: 0.9856 - v
al_loss: 0.1170 - val_acc: 0.9646
Epoch 16/20
60000/60000 [==============================] -
 4s 75us/step - loss: 0.0484 - acc: 0.9869 - v
al_loss: 0.1135 - val_acc: 0.9660
Epoch 17/20
```

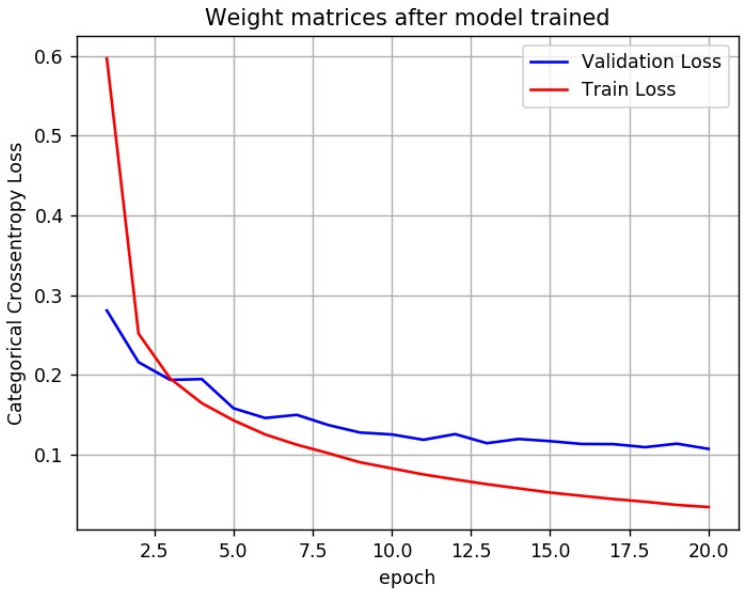
```
60000/60000 [==============================] -
 4s 72us/step - loss: 0.0444 - acc: 0.9880 - v
al_loss: 0.1133 - val_acc: 0.9671
Epoch 18/20
60000/60000 [==============================] -
 4s 71us/step - loss: 0.0411 - acc: 0.9890 - v
al_loss: 0.1095 - val_acc: 0.9684
Epoch 19/20
60000/60000 [==============================] -
 4s 73us/step - loss: 0.0371 - acc: 0.9906 - v
al_loss: 0.1139 - val_acc: 0.9671
Epoch 20/20
60000/60000 [==============================] -
 4s 73us/step - loss: 0.0344 - acc: 0.9914 - v
al_loss: 0.1073 - val_acc: 0.9679
```

In [99]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

```
Test score: 0.1073460508538934
Test accuracy: 0.9679
```

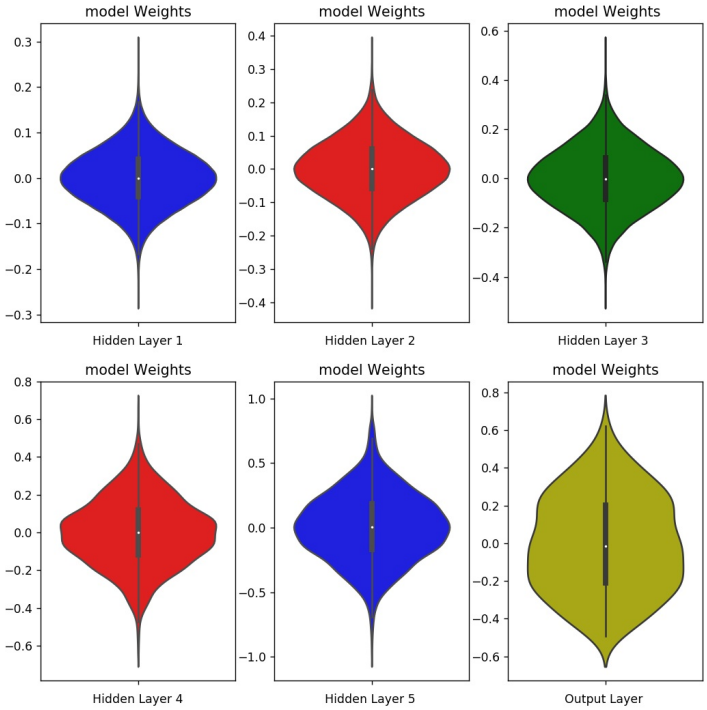


In [100]:

```
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)

plt.title("Weight matrices after model trained")
fig = plt.figure(figsize=(10,10))


plt.subplot(2, 3, 1)
plt.title("model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

plt.subplot(2, 3, 3)
plt.title(" model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(2, 3, 4)
plt.title("model Weights")
ax = sns.violinplot(y=h4_w,color='r')
plt.xlabel('Hidden Layer 4')

plt.subplot(2, 3, 5)
plt.title("model Weights")
ax = sns.violinplot(y=h5_w, color='b')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(2, 3, 6)
plt.title("model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
```

```
plt.show()
```


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

MLP + ReLU + ADAM

In [101]:

```
model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(inp
ut_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.
062, seed=None)))
model_relu.add(Dense(256, activation='relu', kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.088, seed=None)) )
model_relu.add(Dense(128, activation='relu', kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.125, seed=None)) )
model_relu.add(Dense(64, activation='relu', kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.176, seed=None)) )
model_relu.add(Dense(32, activation='relu', kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.25, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_s
ize, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_t
est))
```

```
_________________________________________________________
_________________
Layer (type)                 Output Shape
         Param #
=========================================================
==================
dense_111 (Dense)            (None, 512)
```

```
          401920
_________________________________________________________________
dense_112 (Dense)            (None, 256)
          131328
_________________________________________________________________
dense_113 (Dense)            (None, 128)
          32896
_________________________________________________________________
dense_114 (Dense)            (None, 64)
          8256
_________________________________________________________________
dense_115 (Dense)            (None, 32)
          2080
_________________________________________________________________
dense_116 (Dense)            (None, 10)
          330
=================================================================
Total params: 576,810
Trainable params: 576,810
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.2748 - acc: 0.9183 - val_loss: 0.1250 - val_acc: 0.9633
Epoch 2/20
60000/60000 [==============================] -
```

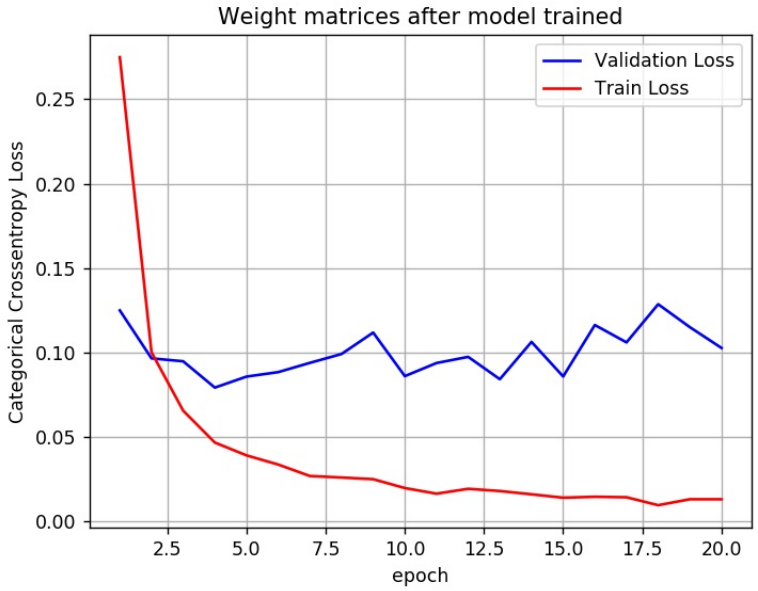
```
 6s 96us/step - loss: 0.1003 - acc: 0.9698 - v
al_loss: 0.0967 - val_acc: 0.9696
Epoch 3/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.0657 - acc: 0.9797 - v
al_loss: 0.0949 - val_acc: 0.9714
Epoch 4/20
60000/60000 [==============================] -
 6s 94us/step - loss: 0.0468 - acc: 0.9853 - v
al_loss: 0.0793 - val_acc: 0.9767
Epoch 5/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.0392 - acc: 0.9875 - v
al_loss: 0.0859 - val_acc: 0.9756
Epoch 6/20
60000/60000 [==============================] -
 6s 100us/step - loss: 0.0338 - acc: 0.9889 -
val_loss: 0.0885 - val_acc: 0.9764
Epoch 7/20
60000/60000 [==============================] -
 6s 101us/step - loss: 0.0271 - acc: 0.9917 -
val_loss: 0.0940 - val_acc: 0.9763
Epoch 8/20
60000/60000 [==============================] -
 6s 103us/step - loss: 0.0262 - acc: 0.9915 -
val_loss: 0.0992 - val_acc: 0.9767
Epoch 9/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0252 - acc: 0.9924 - v
al_loss: 0.1119 - val_acc: 0.9709
Epoch 10/20
60000/60000 [==============================] -
 6s 98us/step - loss: 0.0199 - acc: 0.9935 - v
al_loss: 0.0861 - val_acc: 0.9802
Epoch 11/20
60000/60000 [==============================] -
 6s 103us/step - loss: 0.0166 - acc: 0.9946 -
```

```
val_loss: 0.0939 - val_acc: 0.9772
Epoch 12/20
60000/60000 [==============================] -
 6s 105us/step - loss: 0.0195 - acc: 0.9941 -
val_loss: 0.0975 - val_acc: 0.9764
Epoch 13/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0182 - acc: 0.9941 - v
al_loss: 0.0843 - val_acc: 0.9794
Epoch 14/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.0162 - acc: 0.9950 - v
al_loss: 0.1064 - val_acc: 0.9778
Epoch 15/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.0142 - acc: 0.9957 - v
al_loss: 0.0859 - val_acc: 0.9812
Epoch 16/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0148 - acc: 0.9956 - v
al_loss: 0.1164 - val_acc: 0.9743
Epoch 17/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.0145 - acc: 0.9956 - v
al_loss: 0.1061 - val_acc: 0.9784
Epoch 18/20
60000/60000 [==============================] -
 6s 97us/step - loss: 0.0098 - acc: 0.9968 - v
al_loss: 0.1287 - val_acc: 0.9743
Epoch 19/20
60000/60000 [==============================] -
 6s 99us/step - loss: 0.0133 - acc: 0.9960 - v
al_loss: 0.1151 - val_acc: 0.9762
Epoch 20/20
60000/60000 [==============================] -
 6s 96us/step - loss: 0.0133 - acc: 0.9961 - v
al_loss: 0.1028 - val_acc: 0.9778
```

In [102]:

```
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, validation_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)
```

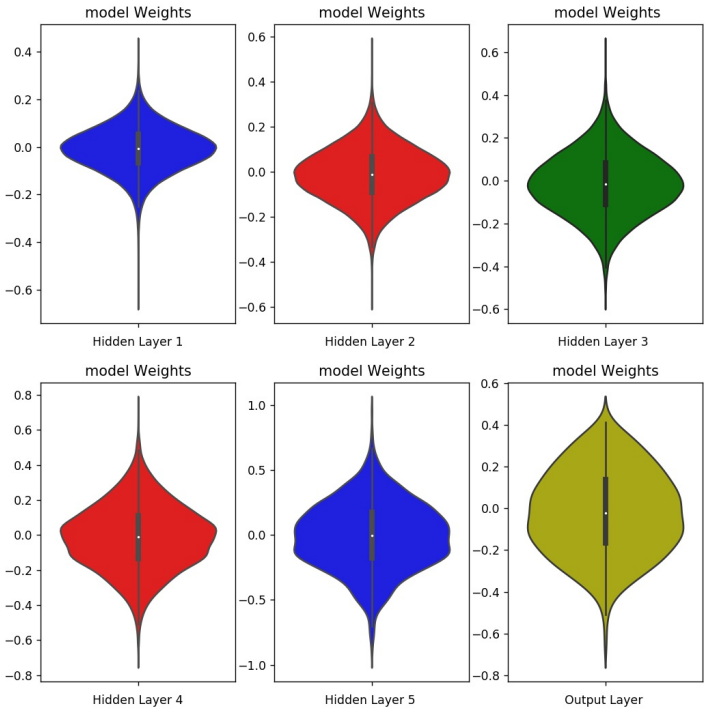
```
Test score: 0.10280663507273835
Test accuracy: 0.9778
```



In [103]:

```
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)

plt.title("Weight matrices after model trained")
fig = plt.figure(figsize=(10,10))


plt.subplot(2, 3, 1)
plt.title("model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(2, 3, 2)
plt.title("model Weights")
```

```
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(2, 3, 3)
plt.title(" model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(2, 3, 4)
plt.title("model Weights")
ax = sns.violinplot(y=h4_w,color='r')
plt.xlabel('Hidden Layer 4')

plt.subplot(2, 3, 5)
plt.title("model Weights")
ax = sns.violinplot(y=h5_w, color='b')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(2, 3, 6)
plt.title("model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```


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

MLP + Batch-Norm on hidden Layers + AdamOptimizer </2>

In [104]:

```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we s
atisfy this condition 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) = 0.051  => N(0,σ) = N(0,0.051)
# h3 =>  σ=√(2/(ni+ni+1) = 0.072  => N(0,σ) = N(0,0.072)
# h4 =>  σ=√(2/(ni+ni+1) = 0.102  => N(0,σ) = N(0,0.102)
# h5 =>  σ=√(2/(ni+ni+1) = 0.144  => N(0,σ) = N(0,0.144)

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, stdde
v=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(256, activation='sigmoid', kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.051, seed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(128, activation='sigmoid', kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.072, seed=None)) )
model_batch.add(BatchNormalization())
```

```
model_batch.add(Dense(64, activation='sigmoid', kernel_initia
lizer=RandomNormal(mean=0.0, stddev=0.102, seed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(32, activation='sigmoid', kernel_initia
lizer=RandomNormal(mean=0.0, stddev=0.144, seed=None)) )
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
_________________________________________________________
_________________
Layer (type)                 Output Shape
         Param #
=========================================================
==================
dense_117 (Dense)            (None, 512)
         401920
_________________________________________________________
_________________
batch_normalization_11 (Batc (None, 512)
         2048
_________________________________________________________
_________________
dense_118 (Dense)            (None, 256)
         131328
_________________________________________________________
_________________
batch_normalization_12 (Batc (None, 256)
         1024
_________________________________________________________
_________________
dense_119 (Dense)            (None, 128)
         32896
```

```
_________________________________________________
_____________________
batch_normalization_13 (Batc (None, 128)
         512
_________________________________________________
_____________________
dense_120 (Dense)            (None, 64)
         8256
_________________________________________________
_____________________
batch_normalization_14 (Batc (None, 64)
         256
_________________________________________________
_____________________
dense_121 (Dense)            (None, 32)
         2080
_________________________________________________
_____________________
batch_normalization_15 (Batc (None, 32)
         128
_________________________________________________
_____________________
dense_122 (Dense)            (None, 10)
         330
=================================================
==================
Total params: 580,778
Trainable params: 578,794
Non-trainable params: 1,984
_________________________________________________
_____________________
```

In [105]:

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

```
history = model_batch.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 samp
les
Epoch 1/20
60000/60000 [==============================] -
 18s 295us/step - loss: 0.2675 - acc: 0.9238 -
 val_loss: 0.1430 - val_acc: 0.9581
Epoch 2/20
60000/60000 [==============================] -
 12s 208us/step - loss: 0.1196 - acc: 0.9640 -
 val_loss: 0.1144 - val_acc: 0.9642
Epoch 3/20
60000/60000 [==============================] -
 13s 213us/step - loss: 0.0864 - acc: 0.9735 -
 val_loss: 0.1112 - val_acc: 0.9659
Epoch 4/20
60000/60000 [==============================] -
 13s 220us/step - loss: 0.0644 - acc: 0.9797 -
 val_loss: 0.0830 - val_acc: 0.9741
Epoch 5/20
60000/60000 [==============================] -
 14s 226us/step - loss: 0.0527 - acc: 0.9833 -
 val_loss: 0.0840 - val_acc: 0.9732
Epoch 6/20
60000/60000 [==============================] -
 13s 220us/step - loss: 0.0457 - acc: 0.9856 -
 val_loss: 0.0887 - val_acc: 0.9721
Epoch 7/20
60000/60000 [==============================] -
 14s 227us/step - loss: 0.0365 - acc: 0.9883 -
 val_loss: 0.0791 - val_acc: 0.9766
Epoch 8/20
60000/60000 [==============================] -
 13s 212us/step - loss: 0.0322 - acc: 0.9893 -
```

```
 val_loss: 0.0875 - val_acc: 0.9757
Epoch 9/20
60000/60000 [==============================] -
 12s 201us/step - loss: 0.0287 - acc: 0.9905 -
 val_loss: 0.0930 - val_acc: 0.9749
Epoch 10/20
60000/60000 [==============================] -
 11s 189us/step - loss: 0.0255 - acc: 0.9915 -
 val_loss: 0.0926 - val_acc: 0.9758
Epoch 11/20
60000/60000 [==============================] -
 13s 218us/step - loss: 0.0277 - acc: 0.9911 -
 val_loss: 0.0770 - val_acc: 0.9809
Epoch 12/20
60000/60000 [==============================] -
 12s 205us/step - loss: 0.0197 - acc: 0.9932 -
 val_loss: 0.0855 - val_acc: 0.9788
Epoch 13/20
60000/60000 [==============================] -
 13s 212us/step - loss: 0.0204 - acc: 0.9932 -
 val_loss: 0.0916 - val_acc: 0.9772
Epoch 14/20
60000/60000 [==============================] -
 13s 213us/step - loss: 0.0178 - acc: 0.9941 -
 val_loss: 0.0915 - val_acc: 0.9756
Epoch 15/20
60000/60000 [==============================] -
 13s 217us/step - loss: 0.0212 - acc: 0.9926 -
 val_loss: 0.0852 - val_acc: 0.9766
Epoch 16/20
60000/60000 [==============================] -
 13s 212us/step - loss: 0.0183 - acc: 0.9940 -
 val_loss: 0.1004 - val_acc: 0.9769
Epoch 17/20
60000/60000 [==============================] -
 12s 200us/step - loss: 0.0154 - acc: 0.9949 -
 val_loss: 0.0848 - val_acc: 0.9779
```

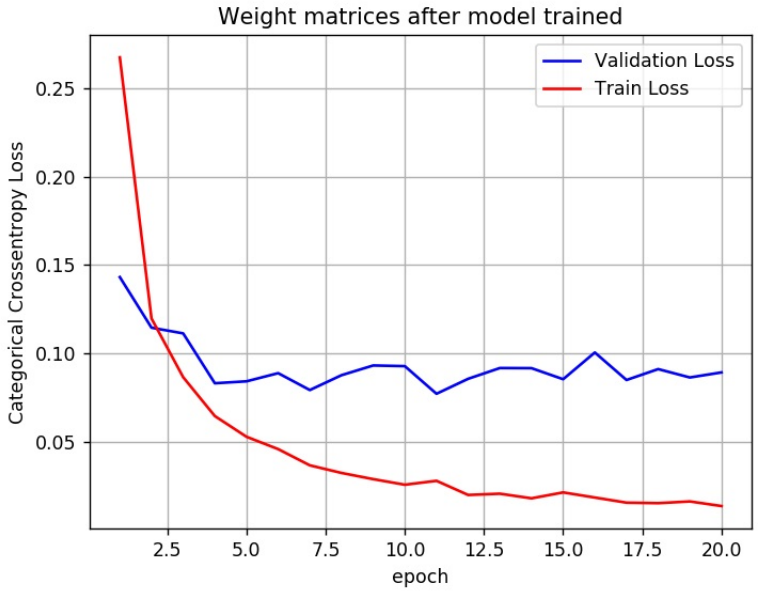
```
Epoch 18/20
60000/60000 [==============================] -
 13s 217us/step - loss: 0.0151 - acc: 0.9950 -
 val_loss: 0.0910 - val_acc: 0.9794
Epoch 19/20
60000/60000 [==============================] -
 13s 213us/step - loss: 0.0160 - acc: 0.9946 -
 val_loss: 0.0862 - val_acc: 0.9791
Epoch 20/20
60000/60000 [==============================] -
 11s 183us/step - loss: 0.0134 - acc: 0.9953 -
 val_loss: 0.0891 - val_acc: 0.9780
```

In [106]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

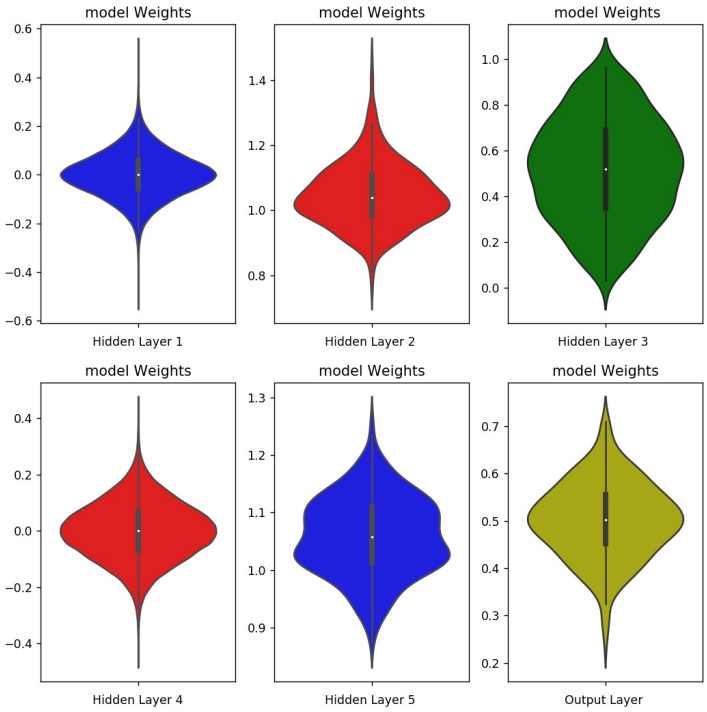
```
Test score: 0.08909254919743981
Test accuracy: 0.978
```



In [107]:

```
w_after = model_batch.get_weights()

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

```
plt.title("Weight matrices after model trained")
fig = plt.figure(figsize=(10,10))


plt.subplot(2, 3, 1)
plt.title("model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

plt.subplot(2, 3, 3)
plt.title(" model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(2, 3, 4)
plt.title("model Weights")
ax = sns.violinplot(y=h4_w,color='r')
plt.xlabel('Hidden Layer 4')

plt.subplot(2, 3, 5)
plt.title("model Weights")
ax = sns.violinplot(y=h5_w, color='b')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(2, 3, 6)
plt.title("model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```


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

5. MLP + Dropout + AdamOptimizer

In [108]:

```
# 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(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(256, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.0, stddev=0.051, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.0, stddev=0.072, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(64, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.0, stddev=0.102, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(32, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.0, stddev=0.144, seed=None)) )
```

```
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
_________________________________________________
_________________
Layer (type)                 Output Shape
          Param #
=================================================
==================
dense_123 (Dense)            (None, 512)
          401920
_________________________________________________
_________________
batch_normalization_16 (Batc (None, 512)
          2048
_________________________________________________
_________________
dropout_6 (Dropout)          (None, 512)
          0
_________________________________________________
_________________
dense_124 (Dense)            (None, 256)
          131328
_________________________________________________
_________________
batch_normalization_17 (Batc (None, 256)
          1024
_________________________________________________
_________________
dropout_7 (Dropout)          (None, 256)
          0
_________________________________________________
```

```
_________________
dense_125 (Dense)            (None, 128)
         32896
_____________________________________________
_________________
batch_normalization_18 (Batc (None, 128)
         512
_____________________________________________
_________________
dropout_8 (Dropout)          (None, 128)
         0
_____________________________________________
_________________
dense_126 (Dense)            (None, 64)
         8256
_____________________________________________
_________________
batch_normalization_19 (Batc (None, 64)
         256
_____________________________________________
_________________
dropout_9 (Dropout)          (None, 64)
         0
_____________________________________________
_________________
dense_127 (Dense)            (None, 32)
         2080
_____________________________________________
_________________
batch_normalization_20 (Batc (None, 32)
         128
_____________________________________________
_________________
dropout_10 (Dropout)         (None, 32)
         0
_____________________________________________
_________________
```

```
dense_128 (Dense)            (None, 10)
         330
=================================================
==================
Total params: 580,778
Trainable params: 578,794
Non-trainable params: 1,984
_________________________________________________
_________________
```

In [109]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_s
ize, epochs=nb_epoch, verbose=1, validation_data=(X_test, Y_t
est))
```

```
Train on 60000 samples, validate on 10000 samp
les
Epoch 1/20
60000/60000 [==============================] -
 12s 193us/step - loss: 1.0613 - acc: 0.6596 -
 val_loss: 0.3441 - val_acc: 0.9018
Epoch 2/20
60000/60000 [==============================] -
 9s 146us/step - loss: 0.6044 - acc: 0.8298 -
val_loss: 0.2848 - val_acc: 0.9183
Epoch 3/20
60000/60000 [==============================] -
 9s 147us/step - loss: 0.5054 - acc: 0.8617 -
val_loss: 0.2638 - val_acc: 0.9269
Epoch 4/20
60000/60000 [==============================] -
 9s 151us/step - loss: 0.4489 - acc: 0.8819 -
val_loss: 0.2435 - val_acc: 0.9322
Epoch 5/20
```

```
60000/60000 [==============================] -
 9s 155us/step - loss: 0.3990 - acc: 0.8964 -
val_loss: 0.2117 - val_acc: 0.9418
Epoch 6/20
60000/60000 [==============================] -
 9s 149us/step - loss: 0.3553 - acc: 0.9095 -
val_loss: 0.1913 - val_acc: 0.9503
Epoch 7/20
60000/60000 [==============================] -
 9s 147us/step - loss: 0.3234 - acc: 0.9182 -
val_loss: 0.1709 - val_acc: 0.9562
Epoch 8/20
60000/60000 [==============================] -
 9s 147us/step - loss: 0.2942 - acc: 0.9258 -
val_loss: 0.1582 - val_acc: 0.9599
Epoch 9/20
60000/60000 [==============================] -
 9s 148us/step - loss: 0.2726 - acc: 0.9328 -
val_loss: 0.1441 - val_acc: 0.9645
Epoch 10/20
60000/60000 [==============================] -
 9s 147us/step - loss: 0.2586 - acc: 0.9378 -
val_loss: 0.1411 - val_acc: 0.9665
Epoch 11/20
60000/60000 [==============================] -
 9s 148us/step - loss: 0.2385 - acc: 0.9432 -
val_loss: 0.1312 - val_acc: 0.9689
Epoch 12/20
60000/60000 [==============================] -
 9s 148us/step - loss: 0.2347 - acc: 0.9444 -
val_loss: 0.1324 - val_acc: 0.9695
Epoch 13/20
60000/60000 [==============================] -
 9s 146us/step - loss: 0.2170 - acc: 0.9475 -
val_loss: 0.1232 - val_acc: 0.9721
Epoch 14/20
60000/60000 [==============================] -
```

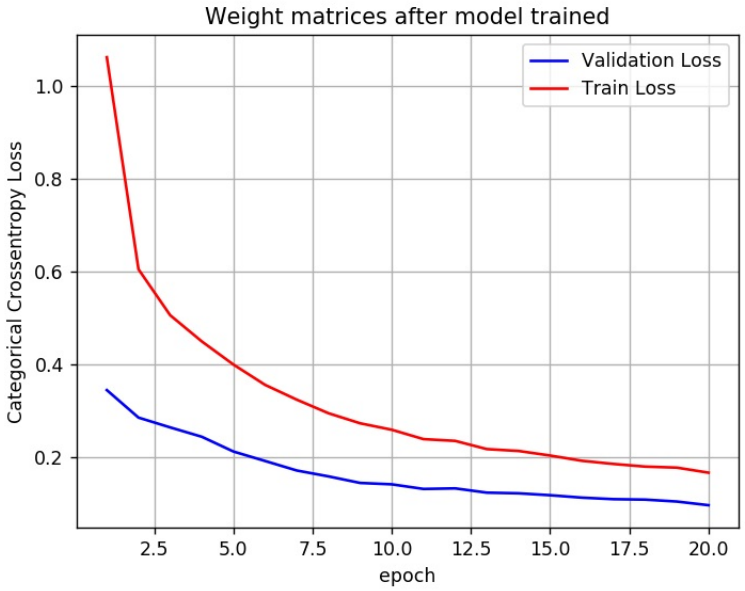
```
 9s 150us/step - loss: 0.2130 - acc: 0.9496 -
val_loss: 0.1218 - val_acc: 0.9735
Epoch 15/20
60000/60000 [==============================] -
 9s 153us/step - loss: 0.2031 - acc: 0.9531 -
val_loss: 0.1176 - val_acc: 0.9741
Epoch 16/20
60000/60000 [==============================] -
 9s 153us/step - loss: 0.1918 - acc: 0.9547 -
val_loss: 0.1124 - val_acc: 0.9744
Epoch 17/20
60000/60000 [==============================] -
 9s 147us/step - loss: 0.1850 - acc: 0.9564 -
val_loss: 0.1091 - val_acc: 0.9766
Epoch 18/20
60000/60000 [==============================] -
 9s 147us/step - loss: 0.1793 - acc: 0.9587 -
val_loss: 0.1082 - val_acc: 0.9772
Epoch 19/20
60000/60000 [==============================] -
 9s 150us/step - loss: 0.1771 - acc: 0.9585 -
val_loss: 0.1040 - val_acc: 0.9776
Epoch 20/20
60000/60000 [==============================] -
 9s 152us/step - loss: 0.1662 - acc: 0.9616 -
val_loss: 0.0961 - val_acc: 0.9804
```

In [110]:

```
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 Crossentr
opy 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, validation_data=(X_test, Y
_test))

# we will get val_loss and val_acc only when you pass the par
amter 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 le
ngth equal to number of epochs

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

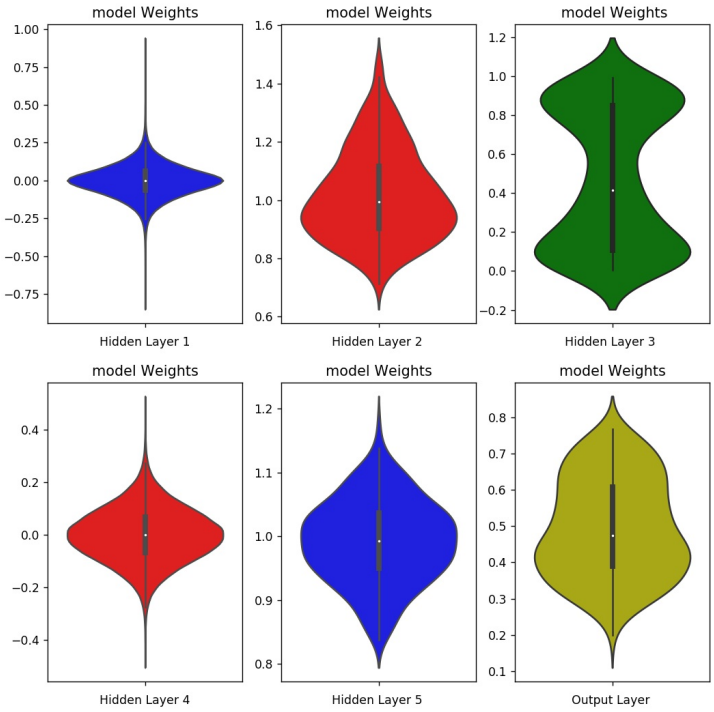
```
Test score: 0.09613309992719442
Test accuracy: 0.9804
```



In [111]:

```
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)
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)

plt.title("Weight matrices after model trained")
fig = plt.figure(figsize=(10,10))


plt.subplot(2, 3, 1)
plt.title("model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(2, 3, 2)
plt.title("model Weights")
```

```
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(2, 3, 3)
plt.title(" model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(2, 3, 4)
plt.title("model Weights")
ax = sns.violinplot(y=h4_w,color='r')
plt.xlabel('Hidden Layer 4')

plt.subplot(2, 3, 5)
plt.title("model Weights")
ax = sns.violinplot(y=h5_w, color='b')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(2, 3, 6)
plt.title("model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```


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

Hyper-parameter tuning of Keras models using Sklearn

In [112]:

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

    model = Sequential()
    model.add(Dense(512, activation=activ, input_shape=(input
_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.06
2, seed=None)))
    model.add(Dense(256, activation=activ, kernel_initializer
=RandomNormal(mean=0.0, stddev=0.088, seed=None)) )
    model.add(Dense(128, activation=activ, kernel_initializer
=RandomNormal(mean=0.0, stddev=0.125, seed=None)) )
    model.add(Dense(64, activation=activ, kernel_initializer=
RandomNormal(mean=0.0, stddev=0.176, seed=None)) )
    model.add(Dense(32, activation=activ, kernel_initializer=
RandomNormal(mean=0.0, stddev=0.25, seed=None)) )
    model.add(Dense(output_dim, activation='softmax'))


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

    return model
```

In [113]:

```
# https://machinelearningmastery.com/grid-search-hyperparamet
ers-deep-learning-models-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_size=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 [114]:

```
print("Best: %f using %s" % (grid_result.best_score_, grid_re
sult.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: 0.974850 using {'activ': 'relu'}
0.971717 (0.000613) with: {'activ': 'sigmoid'}
0.974850 (0.002069) with: {'activ': 'relu'}
```

conclusion

In [116]:

```
# http://zetcode.com/python/prettytable/

from prettytable import PrettyTable

#If you get a ModuleNotFoundError error , install prettytable
 using: pip3 install prettytable

x = PrettyTable()
x.field_names = ["MLP architecture ", "Activation function",
"Optimizer", "Regularization","Train accuracy","Test Accuracy
"]

x.add_row(["Architecture 1", "Sigmoid", "sgd", "None", 0.8705
, 0.8783])
x.add_row(["Architecture 1", "Sigmoid", "Adam", "None", 0.999
4, 0.9828])
x.add_row(["Architecture 1", "ReLu", "sgd", "None", 0.9699, 0
.9630])
x.add_row(["Architecture 1", "ReLu", "Adam", "None", 0.9971,
0.9810])
x.add_row(["Architecture 1", "Sigmoid", "Adam", "Batch Normal
ization", 0.9951, 0.9741])
x.add_row(["Architecture 1", "Sigmoid", "Adam", "Drop out", 0
.9470, 0.9679])


x.add_row(["Architecture 2", "Sigmoid", "sgd", "None", 0.6421
, 0.6652])
x.add_row(["Architecture 2", "Sigmoid", "Adam", "None", 0.998
3, 0.9812])
```

```
x.add_row(["Architecture 2", "ReLu", "sgd", "None", 0.9838, 0
.9685])
x.add_row(["Architecture 2", "ReLu", "Adam", "None", 0.9957,
0.9807])
x.add_row(["Architecture 2", "Sigmoid", "Adam", "Batch Normal
ization", 0.9957, 0.9790])
x.add_row(["Architecture 2", "Sigmoid", "Adam", "Drop out", 0
.9700, 0.9802])


x.add_row(["Architecture 3", "Sigmoid", "sgd", "None", 0.1124
, 0.1135])
x.add_row(["Architecture 3", "Sigmoid", "Adam", "None", 0.996
7, 0.9781])
x.add_row(["Architecture 3", "ReLu", "sgd", "None", 0.9914, 0
.9679])
x.add_row(["Architecture 3", "ReLu", "Adam", "None", 0.9961,
0.9778])
x.add_row(["Architecture 3", "Sigmoid", "Adam", "Batch Normal
ization", 0.9953, 0.9780])
x.add_row(["Architecture 3", "Sigmoid", "Adam", "Drop out", 0
.9616, 0.9804])

print(x)
```

```
+-------------------+---------------------+---
--------+---------------------+--------------
-+---------------+
| MLP architecture  | Activation function | Op
timizer |    Regularization   | Train accuracy
 | Test Accuracy |
+-------------------+---------------------+---
--------+---------------------+--------------
-+---------------+
|   Architecture 1  |       Sigmoid       |
 sgd    |         None        |     0.8705
 |     0.8783    |
```

Architecture 1	Sigmoid	Adam	None	0.9994	0.9828
Architecture 1	ReLu	sgd	None	0.9699	0.963
Architecture 1	ReLu	Adam	None	0.9971	0.981
Architecture 1	Sigmoid	Adam	Batch Normalization	0.9951	0.9741
Architecture 1	Sigmoid	Adam	Drop out	0.947	0.9679
Architecture 2	Sigmoid	sgd	None	0.6421	0.6652
Architecture 2	Sigmoid	Adam	None	0.9983	0.9812
Architecture 2	ReLu	sgd	None	0.9838	0.9685
Architecture 2	ReLu	Adam	None	0.9957	0.9807
Architecture 2	Sigmoid	Adam	Batch Normalization	0.9957	0.979
Architecture 2	Sigmoid	Adam	Drop out	0.97	0.9802
Architecture 3	Sigmoid	sgd	None	0.1124	0.1135
Architecture 3	Sigmoid				

```
 Adam   |          None         |     0.9967
 |     0.9781    |
|   Architecture 3  |         ReLu         |
 sgd    |          None         |     0.9914
 |     0.9679    |
|   Architecture 3  |         ReLu         |
 Adam   |          None         |     0.9961
 |     0.9778    |
|   Architecture 3  |       Sigmoid        |
 Adam   | Batch Normalization |     0.9953
 |     0.978     |
|   Architecture 3  |       Sigmoid        |
 Adam   |       Drop out       |     0.9616
 |     0.9804    |
+-------------------+----------------------+---
--------+----------------------+---------------
-+---------------+
```

Architecture 1 consist of 2 hidden layers with 512 and 128 nuerons respectively. Architecture 2 consist of 3 hidden layers consist of 512, 256 and 64 nuerons respectively. Architecture 3 consist of 5 hidden layers with 512,256,128,64 and 32 nuerons respectively.

conclusion

1.We can observe that as hidden layers increases ,the sgd algorithm with sigmoid activation is performing very bad whereas the sgd optimizer with sigmoid do not change significantly.

2.With increase in hidden layers the execcution time is increasing

3.we do not see significant changes in the train and test accuracy for activation function ReLU and for both optimizers with increase in the hidden layers

4.With increase in the no of hidden layers we do not see significant differences in the train and test accuracies.

5.It would be better to take less no of hidden layers while tuning the DNN network.