MLP Architectures

In [5]:

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

In [6]:

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

In [7]:

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

In [8]:

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

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

In [9]:

```
# if you observe the input shape its 3 dimensional vector
# for each image we have a (28*28) vector
# we will convert the (28*28) vector into single dimensional vector of 1 * 784

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

In [10]:

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

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

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

In [11]:

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

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

In [12]:

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

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

In [13]:

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

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

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

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

In [14]:

```
# 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 [15]:

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

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

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

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

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

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

###

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

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

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

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

####

# https://keras.io/activations/

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

# from keras.layers import Activation, Dense

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

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

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


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

In [16]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [17]:

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

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

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

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

```
WARNING:tensorflow:From C:\Users\Shashank\Anaconda3\lib\site-
```

```
packages\tensorflow\python\framework\op_def_library.py:263: colocate_with (from
tensorflow.python.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

In [18]:

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

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


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

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

# that is why we converted out labels into vectors

model.compile(optimizer='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 function

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

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

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

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

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

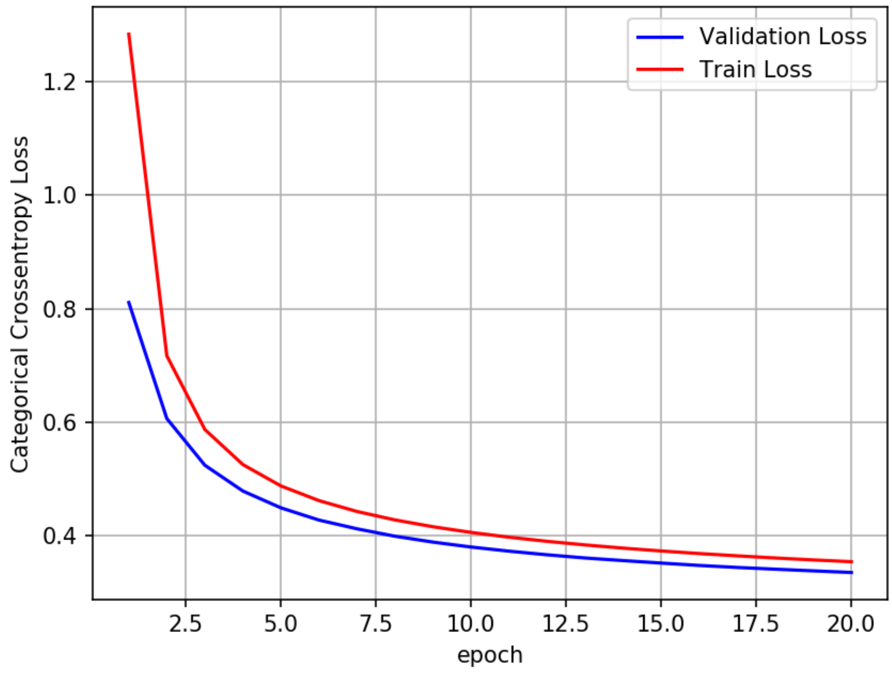
```
WARNING:tensorflow:From C:\Users\Shashank\Anaconda3\lib\site-
packages\tensorflow\python\ops\math_ops.py:3066: to_int32 (from tensorflow.python.ops.math_ops) is
deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 1s 24us/step - loss: 1.2839 - acc: 0.6860 -
val_loss: 0.8109 - val_acc: 0.8381
Epoch 2/20
60000/60000 [==============================] - 1s 20us/step - loss: 0.7170 - acc: 0.8403 -
val_loss: 0.6064 - val_acc: 0.8646
Epoch 3/20
60000/60000 [==============================] - 1s 20us/step - loss: 0.5871 - acc: 0.8588 -
val_loss: 0.5241 - val_acc: 0.8770
Epoch 4/20
60000/60000 [==============================] - 1s 20us/step - loss: 0.5253 - acc: 0.8682 -
val_loss: 0.4787 - val_acc: 0.8830
Epoch 5/20
60000/60000 [==============================] - 1s 21us/step - loss: 0.4876 - acc: 0.8754 -
val_loss: 0.4490 - val_acc: 0.8878
Epoch 6/20
60000/60000 [==============================] - 1s 20us/step - loss: 0.4618 - acc: 0.8801 -
val_loss: 0.4275 - val_acc: 0.8922
Epoch 7/20
60000/60000 [==============================] - 1s 19us/step - loss: 0.4426 - acc: 0.8835 -
```

```
val_loss: 0.4121 - val_acc: 0.8945
Epoch 8/20
60000/60000 [==============================] - 1s 19us/step - loss: 0.4277 - acc: 0.8865 -
val_loss: 0.3990 - val_acc: 0.8968
Epoch 9/20
60000/60000 [==============================] - 1s 20us/step - loss: 0.4157 - acc: 0.8892 -
val_loss: 0.3886 - val_acc: 0.8980
Epoch 10/20
60000/60000 [==============================] - 1s 21us/step - loss: 0.4057 - acc: 0.8916 -
val_loss: 0.3800 - val_acc: 0.8988
Epoch 11/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3973 - acc: 0.8935 -
val_loss: 0.3727 - val_acc: 0.9011
Epoch 12/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3900 - acc: 0.8949 -
val_loss: 0.3663 - val_acc: 0.9026
Epoch 13/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3837 - acc: 0.8964 -
val_loss: 0.3606 - val_acc: 0.9036
Epoch 14/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3780 - acc: 0.8973 -
val_loss: 0.3561 - val_acc: 0.9055
Epoch 15/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.3730 - acc: 0.8986 -
val_loss: 0.3517 - val_acc: 0.9071
Epoch 16/20
60000/60000 [==============================] - 1s 22us/step - loss: 0.3685 - acc: 0.8996 -
val_loss: 0.3475 - val_acc: 0.9078
Epoch 17/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.3644 - acc: 0.9005 -
val_loss: 0.3439 - val_acc: 0.9086
Epoch 18/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.3606 - acc: 0.9014 -
val_loss: 0.3409 - val_acc: 0.9087
Epoch 19/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3572 - acc: 0.9019 -
val_loss: 0.3379 - val_acc: 0.9098
Epoch 20/20
60000/60000 [==============================] - 1s 19us/step - loss: 0.3541 - acc: 0.9025 -
val_loss: 0.3351 - val_acc: 0.9104
```

In [19]:

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

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

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

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

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

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

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

```
Test score: 0.33509153289198873
Test accuracy: 0.9104
```



MLP + Sigmoid activation + SGDOptimizer

In [21]:

```
# Multilayer perceptron

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

model_sigmoid.summary()
```

```
_________________________________________________________________
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
_________________________________________________________________
```

In [22]:

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 126us/step - loss: 2.2614 - acc: 0.2649 -
val_loss: 2.2083 - val_acc: 0.3769
Epoch 2/20
60000/60000 [==============================] - 8s 133us/step - loss: 2.1583 - acc: 0.4630 -
val_loss: 2.0966 - val_acc: 0.4593
Epoch 3/20
60000/60000 [==============================] - 8s 138us/step - loss: 2.0264 - acc: 0.5711 -
val_loss: 1.9356 - val_acc: 0.5871
```

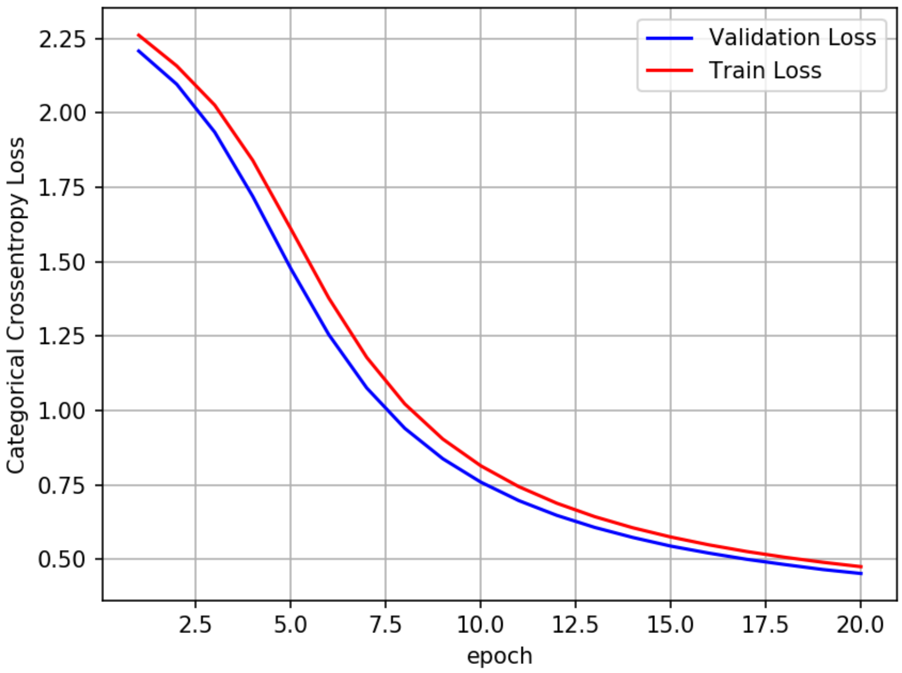
```
Epoch 4/20
60000/60000 [==============================] - 8s 133us/step - loss: 1.8409 - acc: 0.6328 -
val_loss: 1.7199 - val_acc: 0.6586
Epoch 5/20
60000/60000 [==============================] - 8s 141us/step - loss: 1.6112 - acc: 0.6831 -
val_loss: 1.4777 - val_acc: 0.7173
Epoch 6/20
60000/60000 [==============================] - 8s 134us/step - loss: 1.3782 - acc: 0.7293 -
val_loss: 1.2556 - val_acc: 0.7624
Epoch 7/20
60000/60000 [==============================] - 8s 130us/step - loss: 1.1786 - acc: 0.7624 -
val_loss: 1.0760 - val_acc: 0.7861
Epoch 8/20
60000/60000 [==============================] - 8s 130us/step - loss: 1.0225 - acc: 0.7862 -
val_loss: 0.9405 - val_acc: 0.8074
Epoch 9/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.9045 - acc: 0.8033 -
val_loss: 0.8384 - val_acc: 0.8173
Epoch 10/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.8146 - acc: 0.8161 -
val_loss: 0.7597 - val_acc: 0.8311
Epoch 11/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.7445 - acc: 0.8271 -
val_loss: 0.6976 - val_acc: 0.8395
Epoch 12/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.6888 - acc: 0.8359 -
val_loss: 0.6480 - val_acc: 0.8451
Epoch 13/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.6436 - acc: 0.8435 -
val_loss: 0.6074 - val_acc: 0.8512
Epoch 14/20
60000/60000 [==============================] - 10s 165us/step - loss: 0.6063 - acc: 0.8501 - val_l
oss: 0.5738 - val_acc: 0.8578
Epoch 15/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.5753 - acc: 0.8561 -
val_loss: 0.5447 - val_acc: 0.8630
Epoch 16/20
60000/60000 [==============================] - 8s 137us/step - loss: 0.5490 - acc: 0.8601 -
val_loss: 0.5213 - val_acc: 0.8671
Epoch 17/20
60000/60000 [==============================] - 10s 162us/step - loss: 0.5265 - acc: 0.8646 - val_l
oss: 0.5004 - val_acc: 0.8711
Epoch 18/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.5073 - acc: 0.8681 -
val_loss: 0.4827 - val_acc: 0.8742
Epoch 19/20
60000/60000 [==============================] - 8s 128us/step - loss: 0.4904 - acc: 0.8706 -
val_loss: 0.4661 - val_acc: 0.8774
Epoch 20/20
60000/60000 [==============================] - 8s 135us/step - loss: 0.4757 - acc: 0.8739 -
val_loss: 0.4528 - val_acc: 0.8797
```

In [23]:

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

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

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

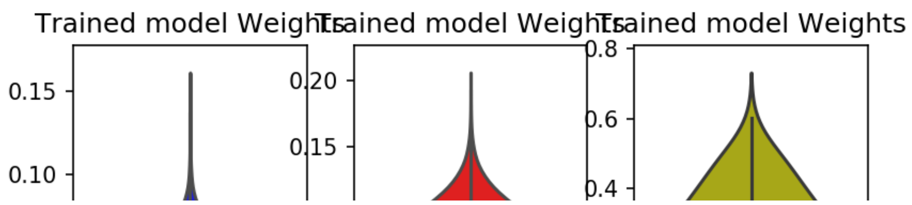
```
vy = history.history['val_loss']
ty = history.history['loss']
plt_dynamic(x, vy, ty, ax)
```

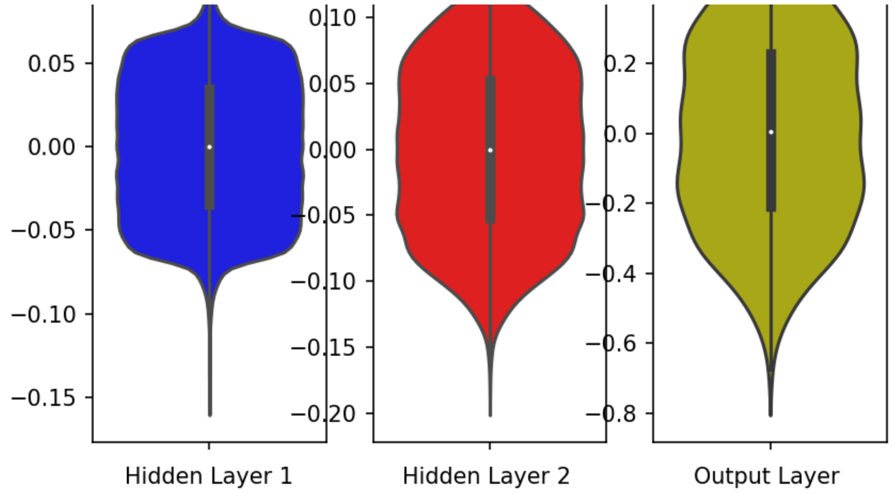
```
Test score: 0.4527508878707886
Test accuracy: 0.8797
```



In [24]:

```
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 [25]:

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

model_sigmoid.summary()

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

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, va
lidation_data=(X_test, Y_test))
```

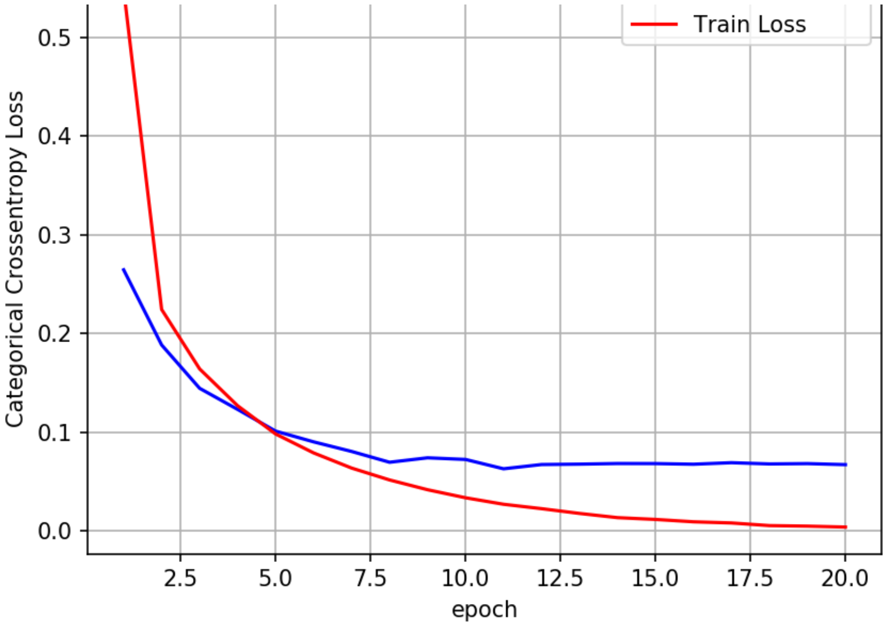
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_8 (Dense)              (None, 512)               401920    
_________________________________________________________________
dense_9 (Dense)              (None, 128)               65664     
_________________________________________________________________
dense_10 (Dense)             (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 12s 202us/step - loss: 0.5493 - acc: 0.8559 - val_l
oss: 0.2645 - val_acc: 0.9198
Epoch 2/20
60000/60000 [==============================] - 11s 180us/step - loss: 0.2243 - acc: 0.9337 - val_l
oss: 0.1883 - val_acc: 0.9448
Epoch 3/20
60000/60000 [==============================] - 10s 173us/step - loss: 0.1640 - acc: 0.9516 - val_l
oss: 0.1444 - val_acc: 0.9579
Epoch 4/20
60000/60000 [==============================] - 12s 192us/step - loss: 0.1267 - acc: 0.9626 - val_l
oss: 0.1229 - val_acc: 0.9623: 0.1264 - acc: 0.
Epoch 5/20
60000/60000 [==============================] - 12s 205us/step - loss: 0.0980 - acc: 0.9711 - val_l
oss: 0.1009 - val_acc: 0.9676
Epoch 6/20
60000/60000 [==============================] - 12s 196us/step - loss: 0.0788 - acc: 0.9763 - val_l
oss: 0.0900 - val_acc: 0.9729
Epoch 7/20
60000/60000 [==============================] - 12s 199us/step - loss: 0.0635 - acc: 0.9811 - val_l
oss: 0.0803 - val_acc: 0.9750
Epoch 8/20
60000/60000 [==============================] - 12s 207us/step - loss: 0.0514 - acc: 0.9849 - val_l
oss: 0.0693 - val_acc: 0.9783
```

```
Epoch 9/20
60000/60000 [==============================] - 11s 192us/step - loss: 0.0415 - acc: 0.9881 - val_l
oss: 0.0739 - val_acc: 0.9775
Epoch 10/20
60000/60000 [==============================] - 12s 193us/step - loss: 0.0333 - acc: 0.9905 - val_l
oss: 0.0722 - val_acc: 0.9774
Epoch 11/20
60000/60000 [==============================] - 12s 196us/step - loss: 0.0267 - acc: 0.9931 - val_l
oss: 0.0627 - val_acc: 0.9805
Epoch 12/20
60000/60000 [==============================] - 12s 207us/step - loss: 0.0223 - acc: 0.9942 - val_l
oss: 0.0670 - val_acc: 0.9784
Epoch 13/20
60000/60000 [==============================] - 12s 201us/step - loss: 0.0174 - acc: 0.9957 - val_l
oss: 0.0674 - val_acc: 0.9798
Epoch 14/20
60000/60000 [==============================] - 12s 196us/step - loss: 0.0132 - acc: 0.9972 - val_l
oss: 0.0681 - val_acc: 0.9790 - los
Epoch 15/20
60000/60000 [==============================] - 12s 200us/step - loss: 0.0113 - acc: 0.9973 - val_l
oss: 0.0680 - val_acc: 0.9802
Epoch 16/20
60000/60000 [==============================] - 13s 215us/step - loss: 0.0089 - acc: 0.9981 - val_l
oss: 0.0673 - val_acc: 0.9815
Epoch 17/20
60000/60000 [==============================] - 14s 226us/step - loss: 0.0078 - acc: 0.9981 - val_l
oss: 0.0689 - val_acc: 0.9802
Epoch 18/20
60000/60000 [==============================] - 14s 225us/step - loss: 0.0051 - acc: 0.9991 - val_l
oss: 0.0676 - val_acc: 0.9809
Epoch 19/20
60000/60000 [==============================] - 15s 245us/step - loss: 0.0045 - acc: 0.9993 - val_l
oss: 0.0680 - val_acc: 0.9815acc:
Epoch 20/20
60000/60000 [==============================] - 13s 215us/step - loss: 0.0036 - acc: 0.9992 - val_l
oss: 0.0669 - val_acc: 0.9826s - loss: 0.0037 - a
```

In [26]:

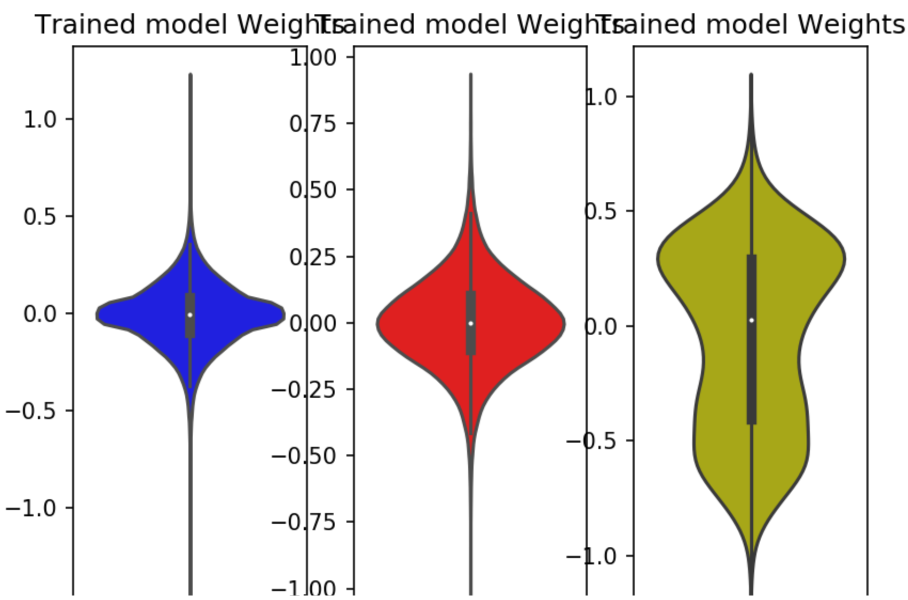
```python
score = model_sigmoid.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, va
lidation_data=(X_test, Y_test))

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

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

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

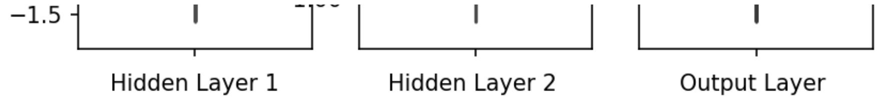
```
Test score: 0.06692838039992276
Test accuracy: 0.9826
```



In [27]:

```
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 [28]:

```
# Multilayer perceptron

# https://arxiv.org/pdf/1707.09725.pdf#page=95
# for relu layers
# If we sample weights from a normal distribution N(0,σ) we satisfy this 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=(input_dim,), kernel_initializer=RandomNor
mal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125
, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_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
_________________________________________________________________
```

In [29]:

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

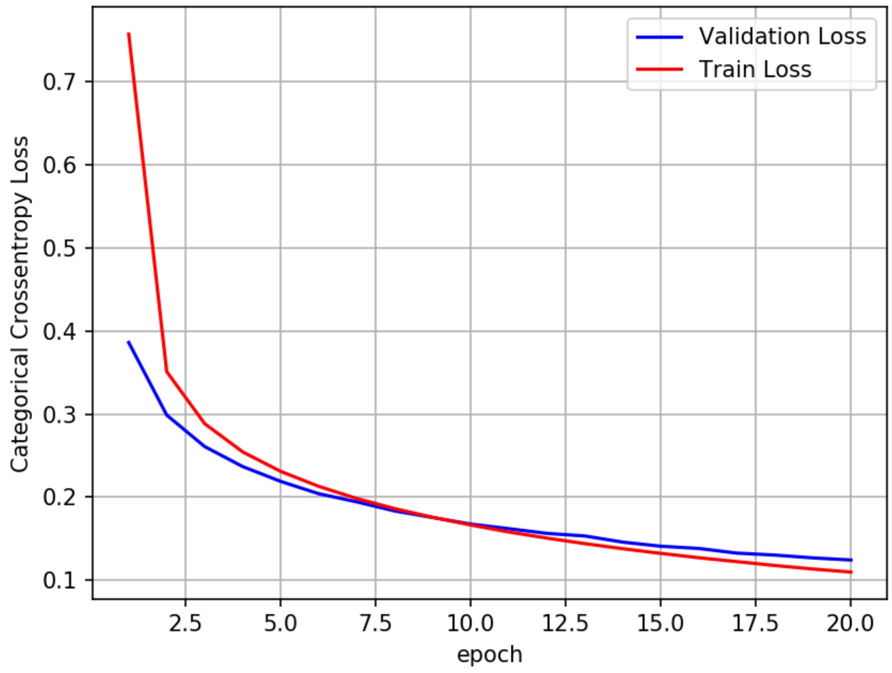
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.7577 - acc: 0.7888 - val_l
oss: 0.3863 - val_acc: 0.8958
Epoch 2/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.3511 - acc: 0.9027 -
val_loss: 0.2988 - val_acc: 0.9164
Epoch 3/20
60000/60000 [==============================] - 8s 135us/step - loss: 0.2885 - acc: 0.9201 -
val_loss: 0.2609 - val_acc: 0.9256
Epoch 4/20
60000/60000 [==============================] - 8s 134us/step - loss: 0.2545 - acc: 0.9292 -
val_loss: 0.2368 - val_acc: 0.9329
Epoch 5/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.2310 - acc: 0.9362 -
val_loss: 0.2190 - val_acc: 0.9367
Epoch 6/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.2130 - acc: 0.9410 -
val_loss: 0.2041 - val_acc: 0.9406
Epoch 7/20
60000/60000 [==============================] - 10s 160us/step - loss: 0.1984 - acc: 0.9447 - val_l
oss: 0.1943 - val_acc: 0.9429
Epoch 8/20
```

```
Epoch 8/20
60000/60000 [==============================] - 8s 125us/step - loss: 0.1862 - acc: 0.9479 -
val_loss: 0.1833 - val_acc: 0.9454
Epoch 9/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.1757 - acc: 0.9510 -
val_loss: 0.1755 - val_acc: 0.9470
Epoch 10/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.1664 - acc: 0.9538 -
val_loss: 0.1676 - val_acc: 0.9504
Epoch 11/20
60000/60000 [==============================] - 8s 136us/step - loss: 0.1580 - acc: 0.9561 -
val_loss: 0.1621 - val_acc: 0.9521
Epoch 12/20
60000/60000 [==============================] - 8s 129us/step - loss: 0.1507 - acc: 0.9582 -
val_loss: 0.1564 - val_acc: 0.9543
Epoch 13/20
60000/60000 [==============================] - 8s 138us/step - loss: 0.1440 - acc: 0.9599 -
val_loss: 0.1533 - val_acc: 0.9551
Epoch 14/20
60000/60000 [==============================] - 9s 148us/step - loss: 0.1379 - acc: 0.9618 -
val_loss: 0.1458 - val_acc: 0.9577
Epoch 15/20
60000/60000 [==============================] - 8s 127us/step - loss: 0.1323 - acc: 0.9632 -
val_loss: 0.1409 - val_acc: 0.9588
Epoch 16/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.1270 - acc: 0.9650 -
val_loss: 0.1382 - val_acc: 0.9603
Epoch 17/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.1224 - acc: 0.9661 -
val_loss: 0.1326 - val_acc: 0.9602
Epoch 18/20
60000/60000 [==============================] - 8s 134us/step - loss: 0.1177 - acc: 0.9676 -
val_loss: 0.1302 - val_acc: 0.9620
Epoch 19/20
60000/60000 [==============================] - 7s 119us/step - loss: 0.1135 - acc: 0.9686 -
val_loss: 0.1268 - val_acc: 0.9634
Epoch 20/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.1098 - acc: 0.9695 -
val_loss: 0.1243 - val_acc: 0.9635
```

In [30]:

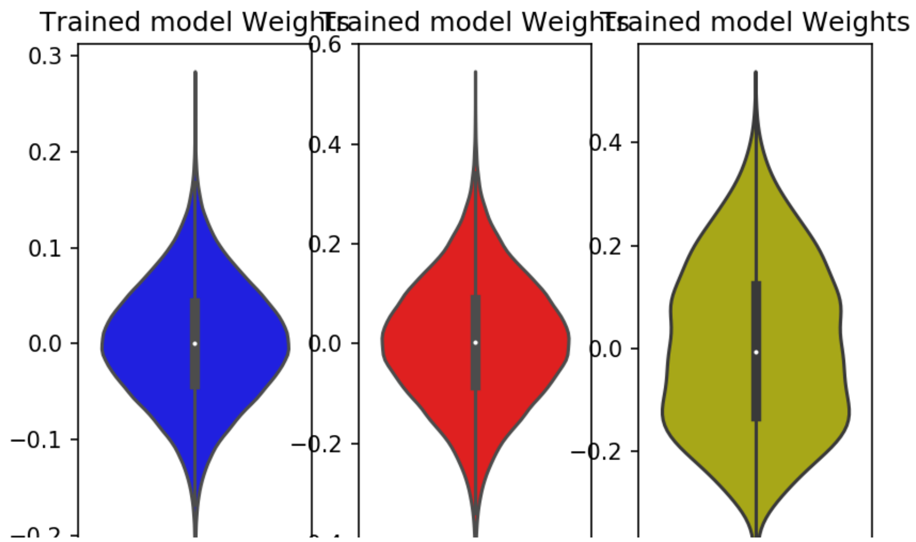
```python
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

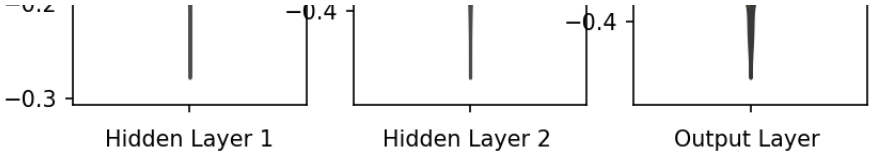
```
Test score: 0.12430026308484375
Test accuracy: 0.9635
```



In [31]:

```
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 [32]:

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

print(model_relu.summary())

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

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

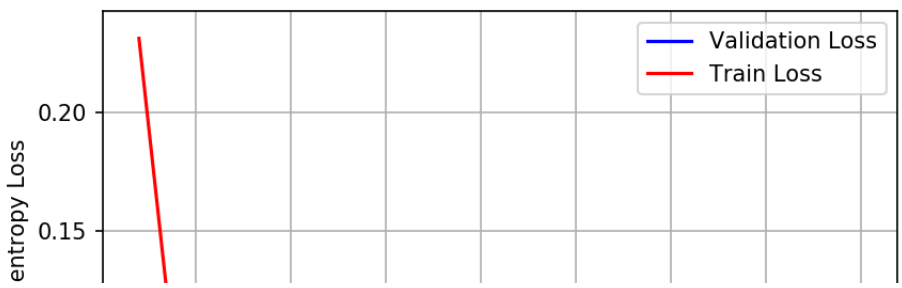
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_14 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_15 (Dense)             (None, 128)               65664
_________________________________________________________________
dense_16 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.2312 - acc: 0.9304 - val_l
oss: 0.1250 - val_acc: 0.9617
Epoch 2/20
60000/60000 [==============================] - 10s 160us/step - loss: 0.0854 - acc: 0.9745 - val_l
oss: 0.0906 - val_acc: 0.9711
Epoch 3/20
60000/60000 [==============================] - 9s 156us/step - loss: 0.0537 - acc: 0.9836 -
val_loss: 0.0872 - val_acc: 0.9728
Epoch 4/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.0362 - acc: 0.9889 - val_l
oss: 0.0671 - val_acc: 0.9790
Epoch 5/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.0262 - acc: 0.9912 - val_l
oss: 0.0690 - val_acc: 0.9775
Epoch 6/20
60000/60000 [==============================] - 10s 172us/step - loss: 0.0202 - acc: 0.9939 - val_l
oss: 0.0705 - val_acc: 0.9782
Epoch 7/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.0153 - acc: 0.9948 -
val_loss: 0.0863 - val_acc: 0.9764
Epoch 8/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0145 - acc: 0.9951 - val_l
oss: 0.0897 - val_acc: 0.9767
Epoch 9/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.0148 - acc: 0.9948 - val_l
oss: 0.0759 - val_acc: 0.9792
Epoch 10/20
60000/60000 [==============================] - 10s 171us/step - loss: 0.0115 - acc: 0.9962 - val_l
oss: 0.0921 - val_acc: 0.9767
Epoch 11/20
60000/60000 [==============================] - 10s 167us/step - loss: 0.0093 - acc: 0.9970 - val_l
oss: 0.1128 - val_acc: 0.9744
```

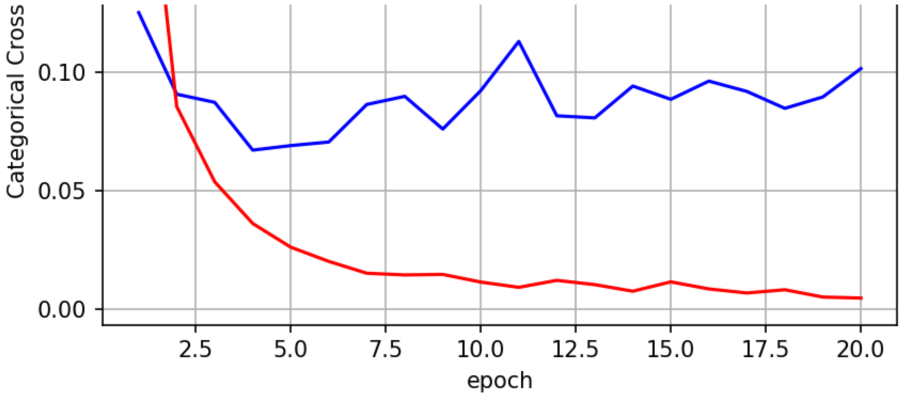
```
Epoch 12/20
60000/60000 [==============================] - 10s 166us/step - loss: 0.0123 - acc: 0.9957 - val_l
oss: 0.0815 - val_acc: 0.9796
Epoch 13/20
60000/60000 [==============================] - 11s 178us/step - loss: 0.0105 - acc: 0.9967 - val_l
oss: 0.0806 - val_acc: 0.9794
Epoch 14/20
60000/60000 [==============================] - 11s 183us/step - loss: 0.0077 - acc: 0.9976 - val_l
oss: 0.0940 - val_acc: 0.9785
Epoch 15/20
60000/60000 [==============================] - 10s 169us/step - loss: 0.0116 - acc: 0.9962 - val_l
oss: 0.0885 - val_acc: 0.9801.0115 - acc: 0
Epoch 16/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0086 - acc: 0.9973 - val_l
oss: 0.0961 - val_acc: 0.9789
Epoch 17/20
60000/60000 [==============================] - 11s 184us/step - loss: 0.0069 - acc: 0.9977 - val_l
oss: 0.0918 - val_acc: 0.9802
Epoch 18/20
60000/60000 [==============================] - 10s 170us/step - loss: 0.0083 - acc: 0.9972 - val_l
oss: 0.0847 - val_acc: 0.9825
Epoch 19/20
60000/60000 [==============================] - 11s 186us/step - loss: 0.0052 - acc: 0.9983 - val_l
oss: 0.0894 - val_acc: 0.9813
Epoch 20/20
60000/60000 [==============================] - 11s 177us/step - loss: 0.0048 - acc: 0.9984 - val_l
oss: 0.1014 - val_acc: 0.9780
```

In [33]:

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

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

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

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

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

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


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

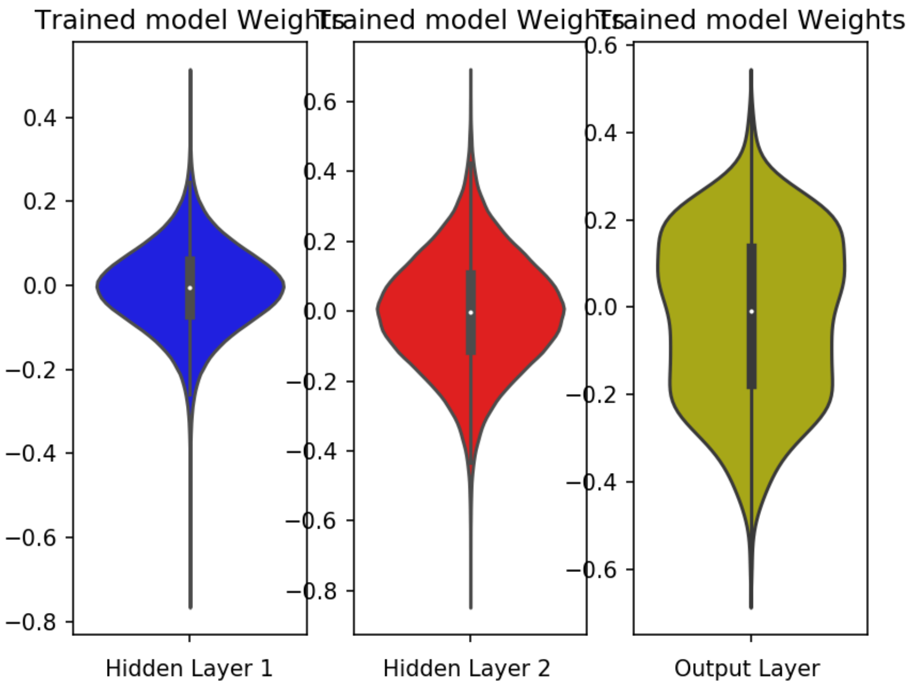
```
Test score: 0.10140154234402549
Test accuracy: 0.978
```





In [34]:

```
w_after = model_relu.get_weights()

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


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

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

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



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

In [35]:

```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this condition with
σ=√(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=Rando
mNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(128, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.0, stddev=0
.55, seed=None)) )
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_17 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048      
_________________________________________________________________
dense_18 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512       
_________________________________________________________________
dense_19 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [36]:

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

history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, vali
dation_data=(X_test, Y_test))
```

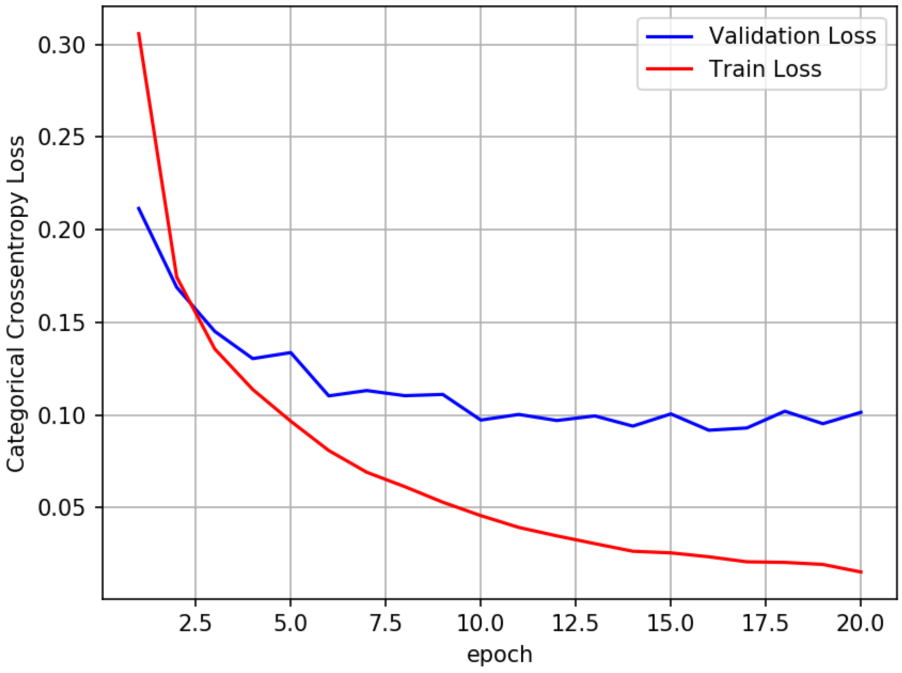
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 14s 228us/step - loss: 0.3058 - acc: 0.9084 - val_l
oss: 0.2115 - val_acc: 0.9404
Epoch 2/20
60000/60000 [==============================] - 12s 194us/step - loss: 0.1744 - acc: 0.9489 - val_l
oss: 0.1689 - val_acc: 0.9510
Epoch 3/20
60000/60000 [==============================] - 12s 203us/step - loss: 0.1357 - acc: 0.9607 - val_l
oss: 0.1452 - val_acc: 0.9575
Epoch 4/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.1138 - acc: 0.9664 - val_l
oss: 0.1304 - val_acc: 0.9623
Epoch 5/20
60000/60000 [==============================] - 12s 193us/step - loss: 0.0966 - acc: 0.9715 - val_l
oss: 0.1337 - val_acc: 0.9596
Epoch 6/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.0808 - acc: 0.9755 - val_l
oss: 0.1103 - val_acc: 0.9667
```

```
Epoch 7/20
60000/60000 [==============================] - 11s 191us/step - loss: 0.0691 - acc: 0.9788 - val_l
oss: 0.1132 - val_acc: 0.9652
Epoch 8/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.0613 - acc: 0.9809 - val_l
oss: 0.1104 - val_acc: 0.9659
Epoch 9/20
60000/60000 [==============================] - 12s 196us/step - loss: 0.0529 - acc: 0.9832 - val_l
oss: 0.1111 - val_acc: 0.9670
Epoch 10/20
60000/60000 [==============================] - 11s 185us/step - loss: 0.0457 - acc: 0.9860 - val_l
oss: 0.0973 - val_acc: 0.9706
Epoch 11/20
60000/60000 [==============================] - 11s 176us/step - loss: 0.0393 - acc: 0.9877 - val_l
oss: 0.1003 - val_acc: 0.9710
Epoch 12/20
60000/60000 [==============================] - 11s 175us/step - loss: 0.0348 - acc: 0.9891 - val_l
oss: 0.0970 - val_acc: 0.9718
Epoch 13/20
60000/60000 [==============================] - 10s 173us/step - loss: 0.0306 - acc: 0.9907 - val_l
oss: 0.0995 - val_acc: 0.9707
Epoch 14/20
60000/60000 [==============================] - 10s 173us/step - loss: 0.0265 - acc: 0.9919 - val_l
oss: 0.0940 - val_acc: 0.9741
Epoch 15/20
60000/60000 [==============================] - 11s 175us/step - loss: 0.0256 - acc: 0.9921 - val_l
oss: 0.1006 - val_acc: 0.9734
Epoch 16/20
60000/60000 [==============================] - 11s 179us/step - loss: 0.0235 - acc: 0.9925 - val_l
oss: 0.0918 - val_acc: 0.9727
Epoch 17/20
60000/60000 [==============================] - 12s 203us/step - loss: 0.0208 - acc: 0.9934 - val_l
oss: 0.0930 - val_acc: 0.9752
Epoch 18/20
60000/60000 [==============================] - 13s 212us/step - loss: 0.0205 - acc: 0.9930 - val_l
oss: 0.1020 - val_acc: 0.9724
Epoch 19/20
60000/60000 [==============================] - 12s 198us/step - loss: 0.0194 - acc: 0.9937 - val_l
oss: 0.0953 - val_acc: 0.9744
Epoch 20/20
60000/60000 [==============================] - 11s 182us/step - loss: 0.0153 - acc: 0.9952 - val_l
oss: 0.1014 - val_acc: 0.9741
```

In [37]:

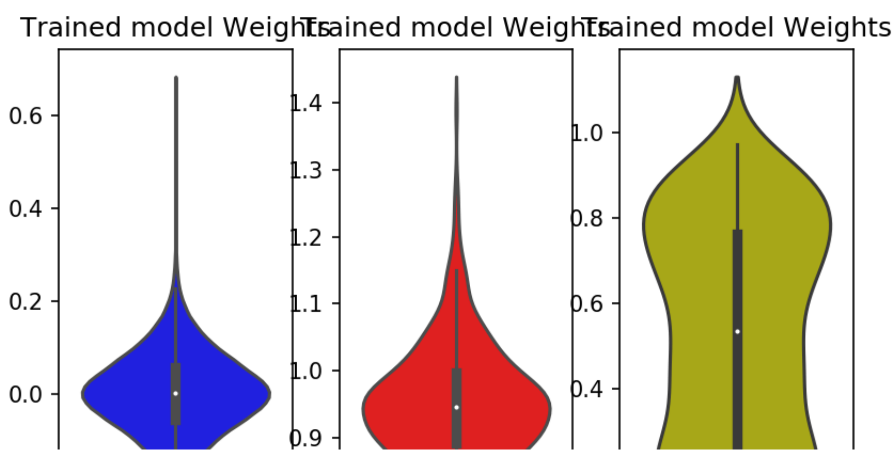
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

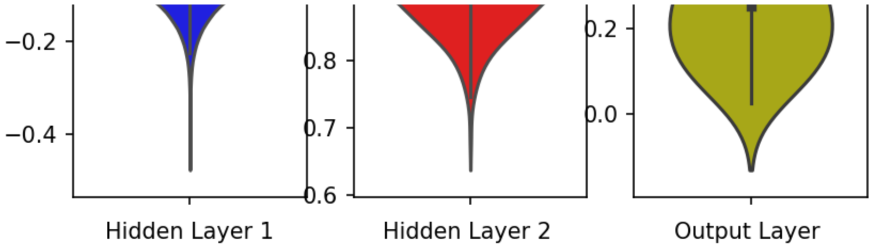
```
Test score: 0.10143673473168165
Test accuracy: 0.9741
```



In [38]:

```
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 [39]:

```
# 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\Shashank\Anaconda3\lib\site-packages\keras\backend\tensorflow_backend.py:3445: calling dropout (from tensorflow.python.ops.nn_ops) with keep_prob is deprecated 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_20 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_21 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_4 (Batch (None, 128)               512       
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_22 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [40]:

```
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))
```

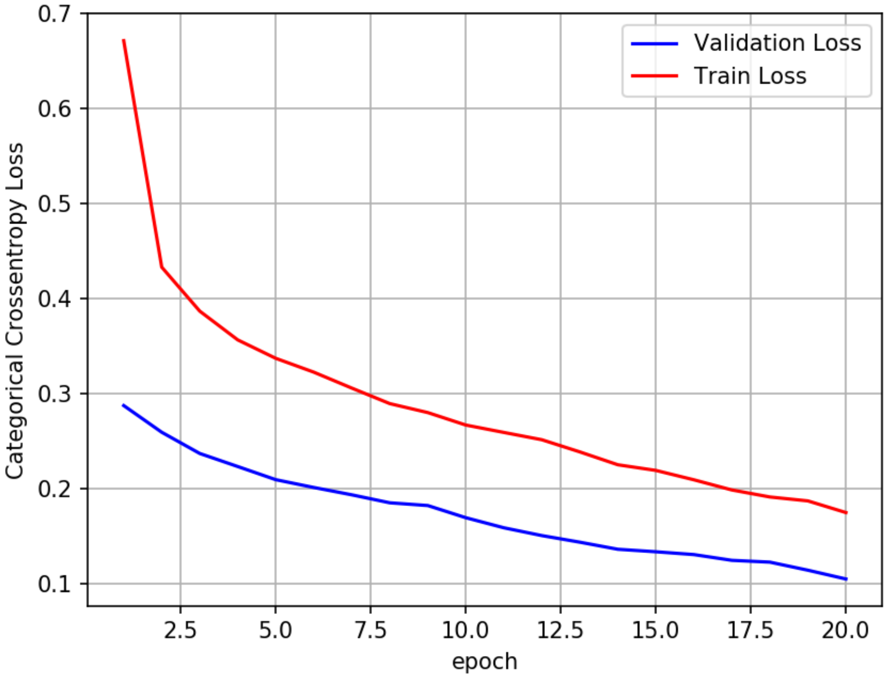
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 14s 225us/step - loss: 0.6715 - acc: 0.7936 - val_l
oss: 0.2873 - val_acc: 0.9132s - los
Epoch 2/20
60000/60000 [==============================] - 12s 201us/step - loss: 0.4331 - acc: 0.8683 - val_l
oss: 0.2592 - val_acc: 0.9231
Epoch 3/20
60000/60000 [==============================] - 12s 192us/step - loss: 0.3868 - acc: 0.8827 - val_l
oss: 0.2369 - val_acc: 0.9335
Epoch 4/20
60000/60000 [==============================] - 11s 187us/step - loss: 0.3565 - acc: 0.8934 - val_l
oss: 0.2231 - val_acc: 0.9332
Epoch 5/20
60000/60000 [==============================] - 11s 188us/step - loss: 0.3371 - acc: 0.8990 - val_l
oss: 0.2093 - val_acc: 0.9389
Epoch 6/20
60000/60000 [==============================] - 12s 202us/step - loss: 0.3225 - acc: 0.9026 - val_l
oss: 0.2011 - val_acc: 0.9393
Epoch 7/20
60000/60000 [==============================] - 13s 213us/step - loss: 0.3057 - acc: 0.9071 - val_l
oss: 0.1934 - val_acc: 0.9413
Epoch 8/20
60000/60000 [==============================] - 12s 204us/step - loss: 0.2894 - acc: 0.9123 - val_l
oss: 0.1850 - val_acc: 0.9458
Epoch 9/20
60000/60000 [==============================] - 12s 203us/step - loss: 0.2799 - acc: 0.9168 - val_l
oss: 0.1821 - val_acc: 0.9461
Epoch 10/20
60000/60000 [==============================] - 12s 200us/step - loss: 0.2668 - acc: 0.9202 - val_l
oss: 0.1693 - val_acc: 0.9504
Epoch 11/20
60000/60000 [==============================] - 12s 194us/step - loss: 0.2591 - acc: 0.9225 - val_l
oss: 0.1587 - val_acc: 0.9528
Epoch 12/20
60000/60000 [==============================] - 12s 208us/step - loss: 0.2514 - acc: 0.9244 - val_l
oss: 0.1504 - val_acc: 0.9562
Epoch 13/20
60000/60000 [==============================] - 13s 220us/step - loss: 0.2385 - acc: 0.9279 - val_l
oss: 0.1435 - val_acc: 0.9564
Epoch 14/20
60000/60000 [==============================] - 13s 212us/step - loss: 0.2250 - acc: 0.9325 - val_l
oss: 0.1360 - val_acc: 0.9602
Epoch 15/20
60000/60000 [==============================] - 12s 197us/step - loss: 0.2190 - acc: 0.9340 - val_l
oss: 0.1334 - val_acc: 0.9591
Epoch 16/20
60000/60000 [==============================] - 14s 235us/step - loss: 0.2092 - acc: 0.9377 - val_l
oss: 0.1304 - val_acc: 0.9601
Epoch 17/20
60000/60000 [==============================] - 12s 196us/step - loss: 0.1985 - acc: 0.9407 - val_l
oss: 0.1244 - val_acc: 0.9651
Epoch 18/20
60000/60000 [==============================] - 12s 195us/step - loss: 0.1911 - acc: 0.9428 - val_l
oss: 0.1225 - val_acc: 0.9635
Epoch 19/20
60000/60000 [==============================] - 13s 224us/step - loss: 0.1870 - acc: 0.9440 - val_l
oss: 0.1140 - val_acc: 0.9672
Epoch 20/20
60000/60000 [==============================] - 13s 222us/step - loss: 0.1747 - acc: 0.9468 - val_l
oss: 0.1047 - val_acc: 0.9699
```

In [41]:

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

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

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

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

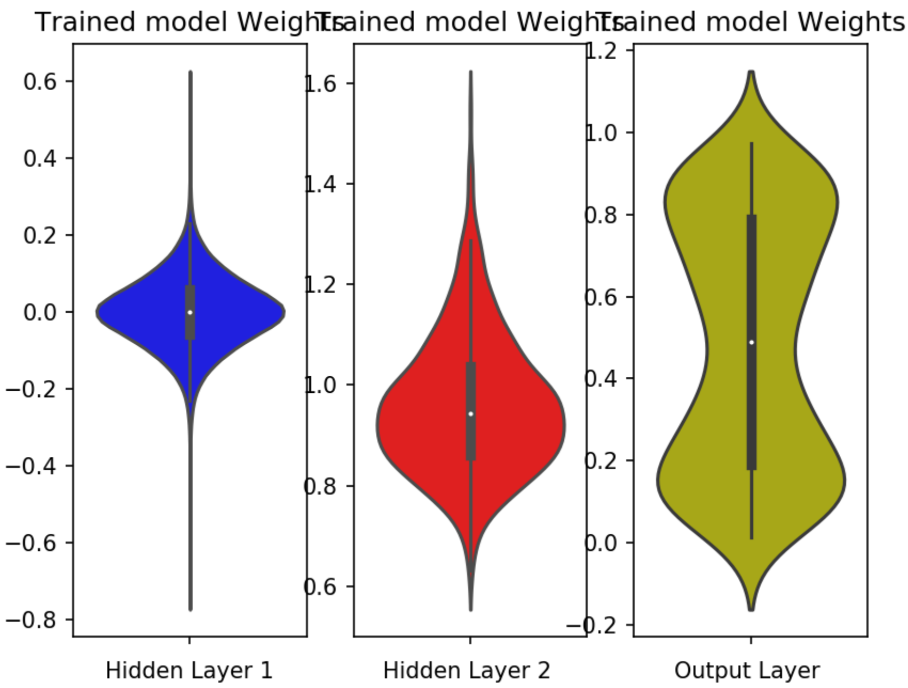
```
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, va
lidation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.10473509188406169
Test accuracy: 0.9699
```



In [42]:

```
w_after = model_drop.get_weights()

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


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

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

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



(MLP + Dropout + AdamOptimizer(with different size of hidden layers(630,410)))

In [43]:

```
# 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(630, activation='sigmoid', input_shape=(input_dim,), kernel_initializer=Random
Normal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(410, 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()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 630)               494550
_________________________________________________________________
batch_normalization_5 (Batch (None, 630)               2520
_________________________________________________________________
dropout_3 (Dropout)          (None, 630)               0
_________________________________________________________________
dense_24 (Dense)             (None, 410)               258710
_________________________________________________________________
batch_normalization_6 (Batch (None, 410)               1640
_________________________________________________________________
dropout_4 (Dropout)          (None, 410)               0
_________________________________________________________________
dense_25 (Dense)             (None, 10)                4110
=================================================================
```

```
Total params: 761,530
Trainable params: 759,450
Non-trainable params: 2,080
_________________________________________________________________
```

In [44]:

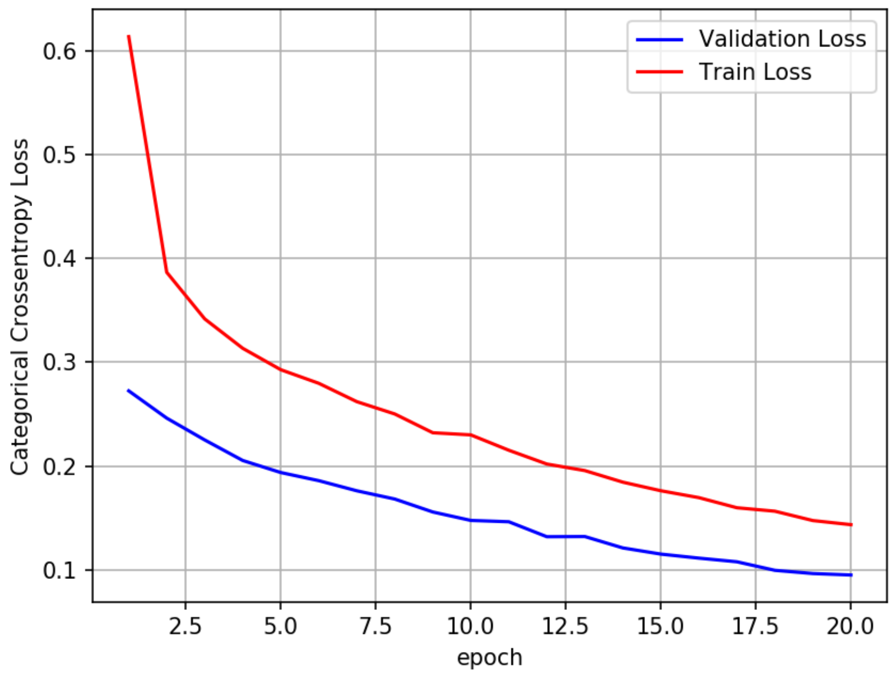
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 24s 398us/step - loss: 0.6136 - acc: 0.8161 - val_l
oss: 0.2725 - val_acc: 0.9234
Epoch 2/20
60000/60000 [==============================] - 22s 360us/step - loss: 0.3867 - acc: 0.8824 - val_l
oss: 0.2462 - val_acc: 0.9301
Epoch 3/20
60000/60000 [==============================] - 20s 327us/step - loss: 0.3418 - acc: 0.8960 - val_l
oss: 0.2252 - val_acc: 0.9343
Epoch 4/20
60000/60000 [==============================] - 23s 379us/step - loss: 0.3135 - acc: 0.9062 - val_l
oss: 0.2054 - val_acc: 0.9393
Epoch 5/20
60000/60000 [==============================] - 21s 346us/step - loss: 0.2929 - acc: 0.9098 - val_l
oss: 0.1938 - val_acc: 0.9432
Epoch 6/20
60000/60000 [==============================] - 20s 336us/step - loss: 0.2798 - acc: 0.9151 - val_l
oss: 0.1860 - val_acc: 0.9465
Epoch 7/20
60000/60000 [==============================] - 18s 302us/step - loss: 0.2621 - acc: 0.9205 - val_l
oss: 0.1762 - val_acc: 0.9483
Epoch 8/20
60000/60000 [==============================] - 18s 292us/step - loss: 0.2502 - acc: 0.9240 - val_l
oss: 0.1683 - val_acc: 0.9501
Epoch 9/20
60000/60000 [==============================] - 18s 300us/step - loss: 0.2321 - acc: 0.9290 - val_l
oss: 0.1558 - val_acc: 0.9555
Epoch 10/20
60000/60000 [==============================] - 18s 294us/step - loss: 0.2301 - acc: 0.9300 - val_l
oss: 0.1477 - val_acc: 0.9567
Epoch 11/20
60000/60000 [==============================] - 19s 318us/step - loss: 0.2153 - acc: 0.9355 - val_l
oss: 0.1464 - val_acc: 0.9582
Epoch 12/20
60000/60000 [==============================] - 24s 393us/step - loss: 0.2020 - acc: 0.9386 - val_l
oss: 0.1320 - val_acc: 0.9622
Epoch 13/20
60000/60000 [==============================] - 19s 318us/step - loss: 0.1957 - acc: 0.9403 - val_l
oss: 0.1322 - val_acc: 0.9629
Epoch 14/20
60000/60000 [==============================] - 19s 310us/step - loss: 0.1845 - acc: 0.9424 - val_l
oss: 0.1212 - val_acc: 0.9642
Epoch 15/20
60000/60000 [==============================] - 19s 313us/step - loss: 0.1763 - acc: 0.9451 - val_l
oss: 0.1153 - val_acc: 0.9674
Epoch 16/20
60000/60000 [==============================] - 19s 314us/step - loss: 0.1696 - acc: 0.9470 - val_l
oss: 0.1114 - val_acc: 0.9680
Epoch 17/20
60000/60000 [==============================] - 26s 429us/step - loss: 0.1599 - acc: 0.9508 - val_l
oss: 0.1079 - val_acc: 0.9680
Epoch 18/20
60000/60000 [==============================] - 26s 426us/step - loss: 0.1566 - acc: 0.9528 - val_l
oss: 0.0996 - val_acc: 0.9713
Epoch 19/20
60000/60000 [==============================] - 29s 487us/step - loss: 0.1476 - acc: 0.9546 - val_l
oss: 0.0965 - val_acc: 0.9732 - acc: 0. - ETA: 0s - loss: 0.1472 - acc: 0.954
Epoch 20/20
60000/60000 [==============================] - 27s 456us/step - loss: 0.1436 - acc: 0.9552 - val_l
oss: 0.0951 - val_acc: 0.9722
```

In [45]:

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

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

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

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

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

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

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

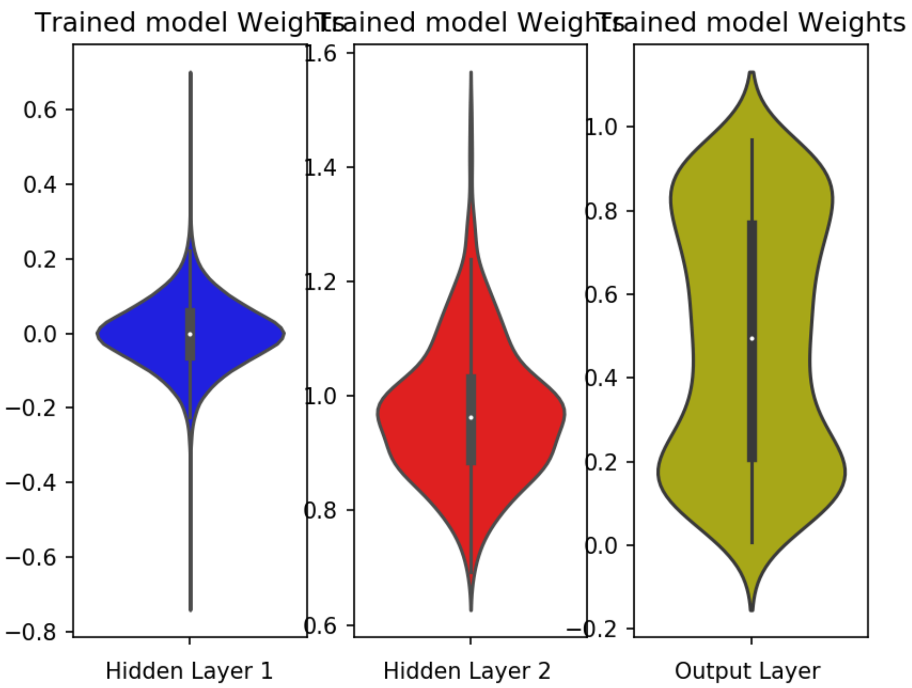
```
Test score: 0.09514947874806821
Test accuracy: 0.9722
```



In [46]:

```
w_after = model_drop.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 3, 1)
plt.title("Trained model Weights")
```

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

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

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



MLP + Dropout + AdamOptimizer(with different size of hidden layers(700,300))

In [47]:

```
# 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(600, 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(200, 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()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
dense_26 (Dense)             (None, 600)               471000
_________________________________________________________________
batch_normalization_7 (Batch (None, 600)               2400
_________________________________________________________________
dropout_5 (Dropout)          (None, 600)               0
_________________________________________________________________
dense_27 (Dense)             (None, 200)               120200
_________________________________________________________________
batch_normalization_8 (Batch (None, 200)               800
_________________________________________________________________
dropout_6 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_28 (Dense)             (None, 10)                2010
=================================================================
Total params: 596,410
Trainable params: 594,810
Non-trainable params: 1,600
_________________________________________________________________
```

In [48]:

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

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

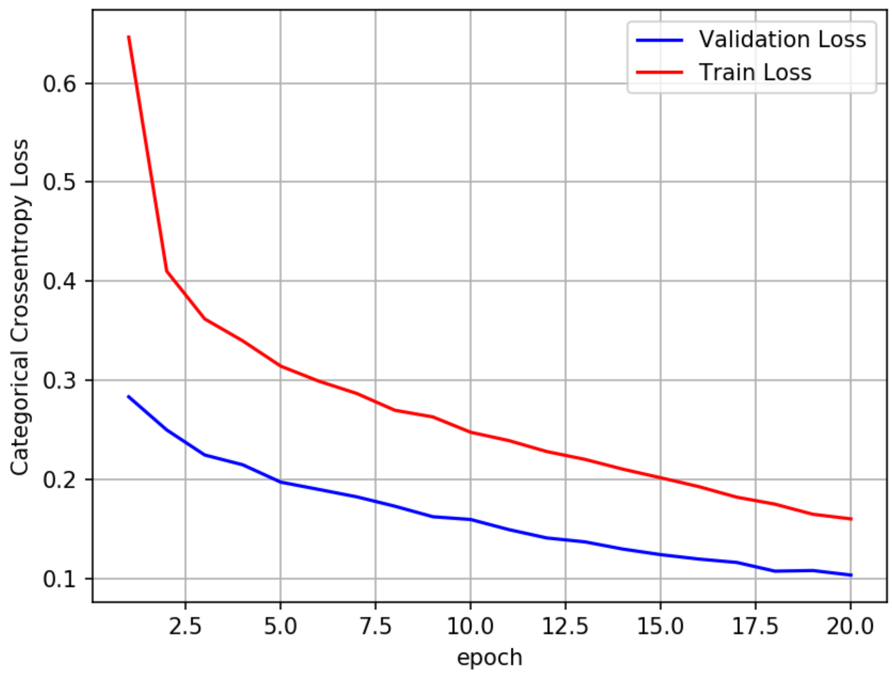
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 22s 374us/step - loss: 0.6463 - acc: 0.8023 - val_l
oss: 0.2833 - val_acc: 0.9171
Epoch 2/20
60000/60000 [==============================] - 20s 333us/step - loss: 0.4101 - acc: 0.8751 - val_l
oss: 0.2499 - val_acc: 0.9264
Epoch 3/20
60000/60000 [==============================] - 21s 342us/step - loss: 0.3619 - acc: 0.8892 - val_l
oss: 0.2246 - val_acc: 0.9344
Epoch 4/20
60000/60000 [==============================] - 19s 316us/step - loss: 0.3398 - acc: 0.8968 - val_l
oss: 0.2147 - val_acc: 0.9355
Epoch 5/20
60000/60000 [==============================] - 17s 283us/step - loss: 0.3143 - acc: 0.9050 - val_l
oss: 0.1972 - val_acc: 0.9407
Epoch 6/20
60000/60000 [==============================] - 20s 327us/step - loss: 0.2991 - acc: 0.9093 - val_l
oss: 0.1898 - val_acc: 0.9421
Epoch 7/20
60000/60000 [==============================] - 19s 309us/step - loss: 0.2867 - acc: 0.9137 - val_l
oss: 0.1823 - val_acc: 0.9467
Epoch 8/20
60000/60000 [==============================] - 21s 357us/step - loss: 0.2698 - acc: 0.9180 - val_l
oss: 0.1729 - val_acc: 0.9493
Epoch 9/20
60000/60000 [==============================] - 21s 354us/step - loss: 0.2630 - acc: 0.9201 - val_l
oss: 0.1623 - val_acc: 0.9530
Epoch 10/20
60000/60000 [==============================] - 20s 334us/step - loss: 0.2475 - acc: 0.9257 - val_l
oss: 0.1595 - val_acc: 0.9512
Epoch 11/20
60000/60000 [==============================] - 19s 315us/step - loss: 0.2392 - acc: 0.9279 - val_l
oss: 0.1493 - val_acc: 0.9560
Epoch 12/20
60000/60000 [==============================] - 19s 316us/step - loss: 0.2280 - acc: 0.9298 - val_l
oss: 0.1408 - val_acc: 0.9582
Epoch 13/20
60000/60000 [==============================] - 20s 338us/step - loss: 0.2202 - acc: 0.9327 - val_l
oss: 0.1369 - val_acc: 0.9593
Epoch 14/20
60000/60000 [==============================] - 18s 307us/step - loss: 0.2103 - acc: 0.9371 - val_l
oss: 0.1296 - val_acc: 0.9606
Epoch 15/20
60000/60000 [==============================] - 16s 268us/step - loss: 0.2016 - acc: 0.9384 - val_l
oss: 0.1240 - val_acc: 0.9620
Epoch 16/20
60000/60000 [==============================] - 14s 240us/step - loss: 0.1926 - acc: 0.9409 - val_l
oss: 0.1196 - val_acc: 0.9642
Epoch 17/20
```

```
Epoch 17/20
60000/60000 [==============================] - 14s 235us/step - loss: 0.1820 - acc: 0.9450 - val_l
oss: 0.1161 - val_acc: 0.9650
Epoch 18/20
60000/60000 [==============================] - 15s 257us/step - loss: 0.1750 - acc: 0.9460 - val_l
oss: 0.1073 - val_acc: 0.9673
Epoch 19/20
60000/60000 [==============================] - 16s 272us/step - loss: 0.1648 - acc: 0.9494 - val_l
oss: 0.1080 - val_acc: 0.9675
Epoch 20/20
60000/60000 [==============================] - 19s 317us/step - loss: 0.1601 - acc: 0.9510 - val_l
oss: 0.1034 - val_acc: 0.9700
```

In [49]:

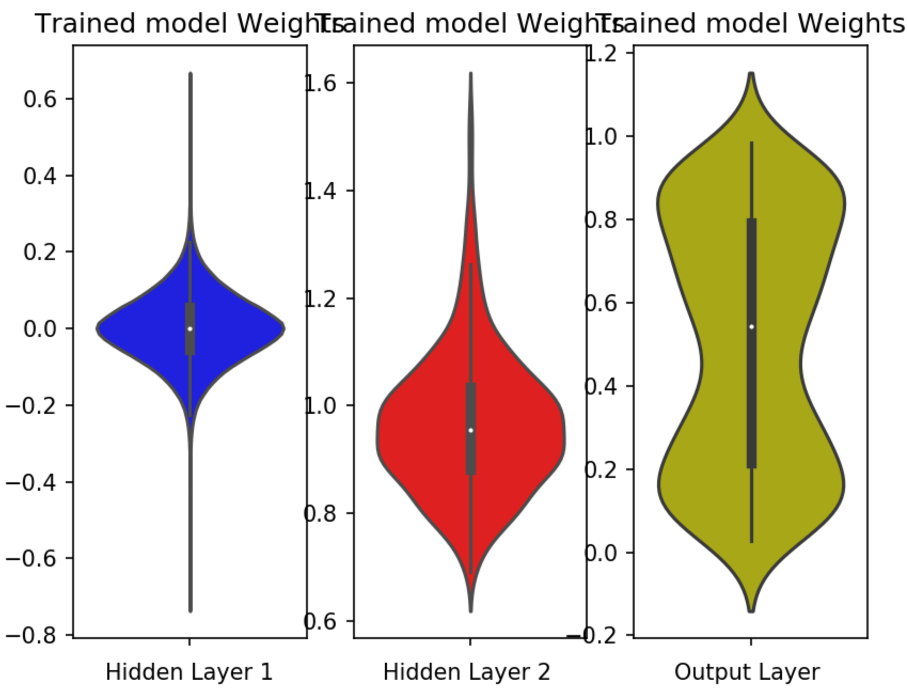
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.10339480265732855
Test accuracy: 0.97
```



In [50]:

```
w_after = model_drop.get_weights()

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


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

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

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



MLP + Dropout + Adam Optimizer with 3 hidden layers(layer1 :800,layer2 :430,layer3:160)

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(800, 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(430, 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(160, 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()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_29 (Dense)             (None, 800)               628000
_________________________________________________________________
batch_normalization_9 (Batch (None, 800)               3200
_________________________________________________________________
dropout_7 (Dropout)          (None, 800)               0
_________________________________________________________________
dense_30 (Dense)             (None, 430)               344430
_________________________________________________________________
batch_normalization_10 (Batc (None, 430)               1720
_________________________________________________________________
dropout_8 (Dropout)          (None, 430)               0
_________________________________________________________________
dense_31 (Dense)             (None, 160)               68960
_________________________________________________________________
batch_normalization_11 (Batc (None, 160)               640
_________________________________________________________________
dropout_9 (Dropout)          (None, 160)               0
_________________________________________________________________
dense_32 (Dense)             (None, 10)                1610
=================================================================
Total params: 1,048,560
Trainable params: 1,045,780
Non-trainable params: 2,780
_________________________________________________________________
```

In [52]:

```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
nb_epoch=15
history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, valid
ation_data=(X_test, Y_test))
```

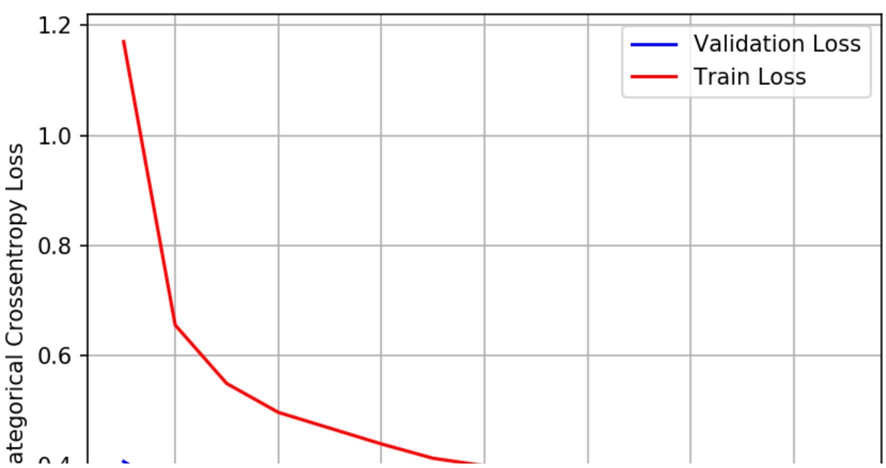
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/15
60000/60000 [==============================] - 34s 570us/step - loss: 1.1707 - acc: 0.6244 - val_l
oss: 0.4072 - val_acc: 0.8815
Epoch 2/15
60000/60000 [==============================] - 27s 454us/step - loss: 0.6552 - acc: 0.7899 - val_l
oss: 0.3481 - val_acc: 0.9004
Epoch 3/15
60000/60000 [==============================] - 31s 524us/step - loss: 0.5489 - acc: 0.8288 - val_l
oss: 0.3212 - val_acc: 0.9074
Epoch 4/15
60000/60000 [==============================] - 35s 591us/step - loss: 0.4965 - acc: 0.8451 - val_l
oss: 0.2922 - val_acc: 0.9145
Epoch 5/15
60000/60000 [==============================] - 33s 542us/step - loss: 0.4678 - acc: 0.8557 - val_l
oss: 0.2880 - val_acc: 0.9176
Epoch 6/15
60000/60000 [==============================] - 33s 558us/step - loss: 0.4391 - acc: 0.8657 - val_l
oss: 0.2738 - val_acc: 0.9211
Epoch 7/15
60000/60000 [==============================] - 33s 558us/step - loss: 0.4131 - acc: 0.8740 - val_l
oss: 0.2632 - val_acc: 0.9238
Epoch 8/15
60000/60000 [==============================] - 30s 495us/step - loss: 0.3994 - acc: 0.8792 - val_l
oss: 0.2498 - val_acc: 0.9286
```

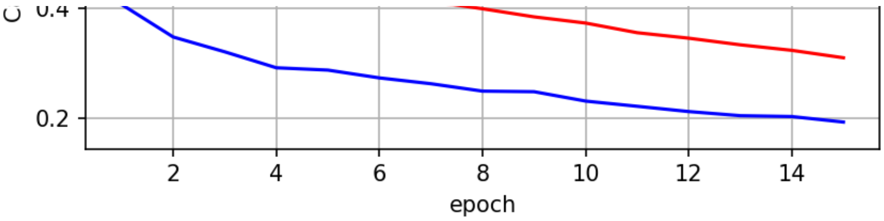
```
oss: 0.2490 - val_acc: 0.9266
Epoch 9/15
60000/60000 [==============================] - 29s 488us/step - loss: 0.3847 - acc: 0.8837 - val_l
oss: 0.2488 - val_acc: 0.9279
Epoch 10/15
60000/60000 [==============================] - 28s 465us/step - loss: 0.3734 - acc: 0.8869 - val_l
oss: 0.2317 - val_acc: 0.9341
Epoch 11/15
60000/60000 [==============================] - 32s 536us/step - loss: 0.3559 - acc: 0.8931 - val_l
oss: 0.2223 - val_acc: 0.9376
Epoch 12/15
60000/60000 [==============================] - 38s 637us/step - loss: 0.3460 - acc: 0.8960 - val_l
oss: 0.2127 - val_acc: 0.9396
Epoch 13/15
60000/60000 [==============================] - 30s 504us/step - loss: 0.3339 - acc: 0.9001 - val_l
oss: 0.2052 - val_acc: 0.9404 - a - ETA: 2
Epoch 14/15
60000/60000 [==============================] - 26s 435us/step - loss: 0.3239 - acc: 0.9036 - val_l
oss: 0.2037 - val_acc: 0.9433
Epoch 15/15
60000/60000 [==============================] - 27s 455us/step - loss: 0.3104 - acc: 0.9085 - val_l
oss: 0.1936 - val_acc: 0.9446
```

In [53]:

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

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

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

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

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

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

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

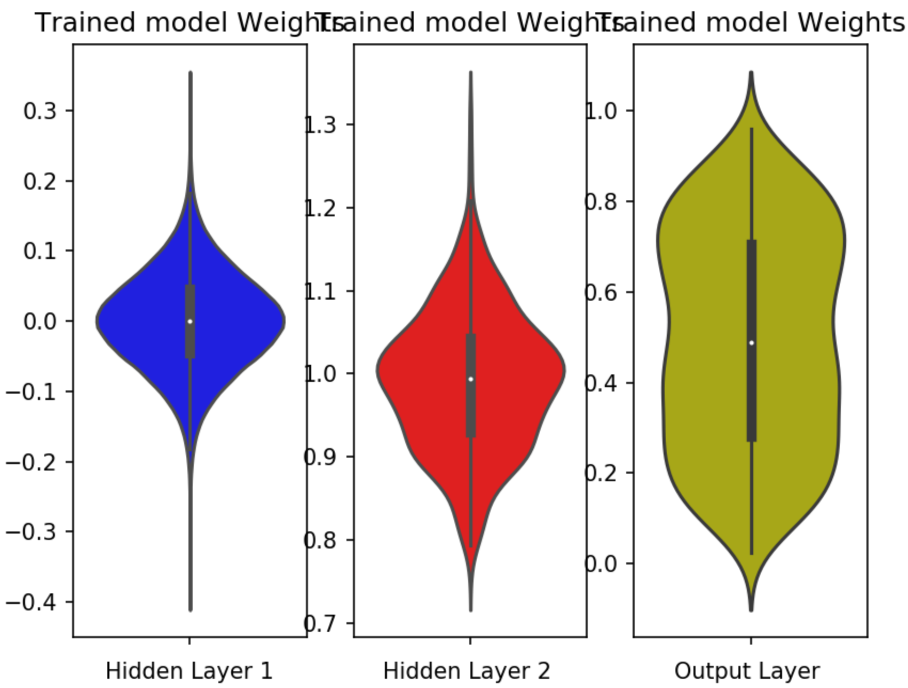
```
Test score: 0.19363341980278492
Test accuracy: 0.9446
```





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()
```



In [60]:

```
# 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 = ["Model","Accuracy"]
x.add_row(["Softmax classifer",90.02])
x.add_row(["MLP+Sigmod activation+SGD Optimizer",88.05])
x.add_row(["MLP+Sigmoid activation+ADAM",97.95])
x.add_row(["MLP+ReLU+SGD",98.46])
x.add_row(["MLP+ReLU+ADAM",98.43])
x.add_row(["MLP+Batch Norm on hidden Layers+Adam Optimizer",97.37])
x.add_row([ "MLP+Dropout+AdamOptimizer",96.92])
x.add_row(["MLP +Dropout + AdamOptimizer(different size of hidden layers(540,360))",97.05])
x.add_row(["MLP +Dropout+AdamOptimizer(with different size of hidden layers(600,200))|96.48",96.48
])

x.add_row(["MLP+Dropout+Adam Optimizer with 3 hidden layers",97.48])
x.add_row(["MLP +Dropout+Adam Optimizer with 3 hidden layers",96.09])

print(x)
```

```
+----------------------------------------------------------------------------------+----------+
|                                      Model                                       | Accuracy |
+----------------------------------------------------------------------------------+----------+
|                                Softmax classifer                                 |  90.02   |
|                       MLP+Sigmod activation+SGD Optimizer                        |  88.05   |
|                           MLP+Sigmoid activation+ADAM                            |  97.95   |
|                                   MLP+ReLU+SGD                                   |  98.46   |
|                                  MLP+ReLU+ADAM                                   |  98.43   |
|                  MLP+Batch Norm on hidden Layers+Adam Optimizer                  |  97.37   |
|                            MLP+Dropout+AdamOptimizer                             |  96.92   |
|      MLP +Dropout + AdamOptimizer(different size of hidden layers(540,360))      |  97.05   |
| MLP +Dropout+AdamOptimizer(with different size of hidden layers(600,200))|96.48 |  96.48   |
|                 MLP+Dropout+Adam Optimizer with 3 hidden layers                  |  97.48   |
|                 MLP +Dropout+Adam Optimizer with 3 hidden layers                 |  96.09   |
+----------------------------------------------------------------------------------+----------+
```

Conclusion

1.On the mnist dataset sigmoid alongwith SGD optimiser is used which does not give much accuracy

2.On changing the optimiser keeping the activation unit same accuracy has increased

3.Again on changing activation unit to Relu and using SGD as an optimiser accuracy has improved

4.The accuracy decreased bit on replacing optimiser with adam

5.On using dropout the accuracy did not increase much,may be because dataset is small

6.On using dropout along with alongwith 2 and 3 layers accuracy has improved.