Keras -- MLPs on MNIST

In [0]:

```
# 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
```

In [0]:

```
%matplotlib inline
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 [4]:

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

```
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [==============================] - 2s 0us/step
```

In [5]:

```
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.s
hape[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 [0]:

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

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

In [7]:

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

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

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

In [7]:

```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 [0]:

```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 [9]:

```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.         0.         0.         0.         0.
0.         0.05490196 0.00392157 0.60392157 0.99215686 0.35294118
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.54509804 0.99215686 0.74509804 0.00784314 0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.04313725
0.74509804 0.99215686 0.2745098  0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.1372549  0.94509804
0.88235294 0.62745098 0.42352941 0.00392157 0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.31764706 0.94117647 0.99215686
0.99215686 0.46666667 0.09803922 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.17647059 0.72941176 0.99215686 0.99215686
0.58823529 0.10588235 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.0627451  0.36470588 0.98823529 0.99215686 0.73333333
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.97647059 0.99215686 0.97647059 0.25098039 0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
```

```
 0.         0.         0.         0.         0.         0.
 0.         0.         0.18039216 0.50980392 0.71764706 0.99215686
 0.99215686 0.81176471 0.00784314 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.15294118 0.58039216
 0.89803922 0.99215686 0.99215686 0.99215686 0.98039216 0.71372549
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.09411765 0.44705882 0.86666667 0.99215686 0.99215686 0.99215686
 0.99215686 0.78823529 0.30588235 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.09019608 0.25882353 0.83529412 0.99215686
 0.99215686 0.99215686 0.99215686 0.77647059 0.31764706 0.00784314
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.07058824 0.67058824
 0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.76470588
 0.31372549 0.03529412 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.21568627 0.6745098  0.88627451 0.99215686 0.99215686 0.99215686
 0.99215686 0.95686275 0.52156863 0.04313725 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.53333333 0.99215686
 0.99215686 0.99215686 0.83137255 0.52941176 0.51764706 0.0627451
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]
```

In [10]:

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

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

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

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

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

Softmax classifier

In [0]:

```
# 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:
```

```python
# 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 suppor
ted 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 [0]:

```python
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [13]:

```python
# 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 /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:66:
The name tf.get_default_graph is deprecated. Please use tf.compat.v1.get_default_graph instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:541:
The name tf.placeholder is deprecated. Please use tf.compat.v1.placeholder instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:4432
: The name tf.random_uniform is deprecated. Please use tf.random.uniform instead.
```

MLP + ReLU + ADAM

In [14]:

```
model_relu = Sequential()
model_relu.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(
mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(72, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, see
d=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, validatio
n_data=(X_test, Y_test))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:4409
: The name tf.random_normal is deprecated. Please use tf.random.normal instead.

Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 364)               285740
_________________________________________________________________
dense_3 (Dense)              (None, 72)                26280
_________________________________________________________________
dense_4 (Dense)              (None, 10)                730
=================================================================
Total params: 312,750
Trainable params: 312,750
Non-trainable params: 0
_________________________________________________________________
None
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/optimizers.py:793: The name tf.tra
in.Optimizer is deprecated. Please use tf.compat.v1.train.Optimizer instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:3576
: The name tf.log is deprecated. Please use tf.math.log instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_grad.py:1250:
add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops) is deprecated and will be
removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.2509 - acc: 0.9257 - val_loss: 0
.1234 - val_acc: 0.9620
Epoch 2/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0985 - acc: 0.9710 - val_loss: 0.
0895 - val_acc: 0.9728
Epoch 3/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0618 - acc: 0.9816 - val_loss: 0.
0827 - val_acc: 0.9757
Epoch 4/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0424 - acc: 0.9871 - val_loss: 0.
0909 - val_acc: 0.9735
Epoch 5/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0311 - acc: 0.9904 - val_loss: 0.
0701 - val_acc: 0.9803
```

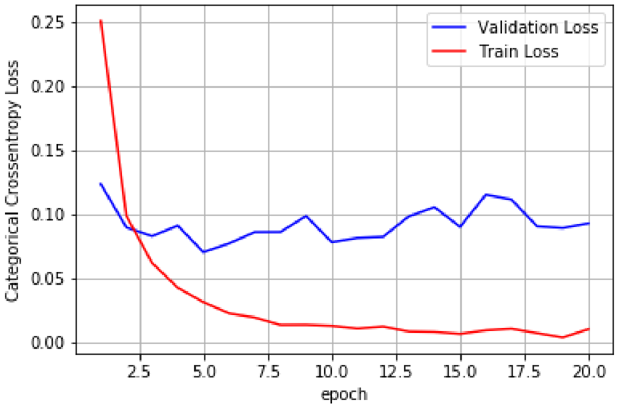
```
0701 - val_acc: 0.9803
Epoch 6/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0223 - acc: 0.9932 - val_loss: 0.
0770 - val_acc: 0.9773
Epoch 7/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0191 - acc: 0.9939 - val_loss: 0.
0859 - val_acc: 0.9771
Epoch 8/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0133 - acc: 0.9959 - val_loss: 0.
0859 - val_acc: 0.9789
Epoch 9/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0133 - acc: 0.9956 - val_loss: 0.
0984 - val_acc: 0.9740
Epoch 10/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0125 - acc: 0.9958 - val_loss: 0.
0779 - val_acc: 0.9807
Epoch 11/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0105 - acc: 0.9966 - val_loss: 0.
0813 - val_acc: 0.9803
Epoch 12/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0120 - acc: 0.9958 - val_loss: 0.
0822 - val_acc: 0.9790
Epoch 13/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0082 - acc: 0.9972 - val_loss: 0.
0981 - val_acc: 0.9770
Epoch 14/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0078 - acc: 0.9976 - val_loss: 0.
1054 - val_acc: 0.9765
Epoch 15/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0063 - acc: 0.9977 - val_loss: 0.
0899 - val_acc: 0.9809
Epoch 16/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.0091 - acc: 0.9971 - val_loss: 0.
1151 - val_acc: 0.9776
Epoch 17/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.0103 - acc: 0.9965 - val_loss: 0.
1113 - val_acc: 0.9785
Epoch 18/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0068 - acc: 0.9976 - val_loss: 0.
0904 - val_acc: 0.9817
Epoch 19/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.0035 - acc: 0.9990 - val_loss: 0.
0891 - val_acc: 0.9813
Epoch 20/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.0100 - acc: 0.9967 - val_loss: 0.
0927 - val_acc: 0.9810
```

In [33]:

```
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(x)

# 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, validat
ion_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']
```

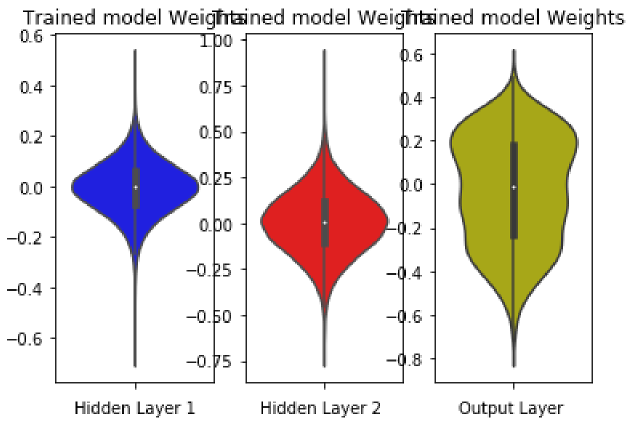
```
#print(ty)
plt_dynamic(x, vy, ty, ax)
```

```
Test score: 0.09268554465530374
Test accuracy: 0.981
```



In [35]:

```
w_after = model_relu.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[2].flatten().reshape(-1,1)
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 [14]:

```
# Multilayer perceptron
```

```
# 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(364, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal
(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(72, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, see
d=None)) )
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:4409
: The name tf.random_normal is deprecated. Please use tf.random.normal instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:148:
The name tf.placeholder_with_default is deprecated. Please use tf.compat.v1.placeholder_with_default in
stead.

Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 364)               285740
_________________________________________________________________
batch_normalization_1 (Batch (None, 364)               1456
_________________________________________________________________
dense_3 (Dense)              (None, 72)                26280
_________________________________________________________________
batch_normalization_2 (Batch (None, 72)                288
_________________________________________________________________
dense_4 (Dense)              (None, 10)                730
=================================================================
Total params: 314,494
Trainable params: 313,622
Non-trainable params: 872
_________________________________________________________________
```

In [15]:

```
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, validati
on_data=(X_test, Y_test))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/optimizers.py:793: The name tf.tra
in.Optimizer is deprecated. Please use tf.compat.v1.train.Optimizer instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_grad.py:1250:
add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops) is deprecated and will be
removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 9s 158us/step - loss: 0.2165 - acc: 0.9376 - val_loss: 0
.1199 - val_acc: 0.9646
Epoch 2/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0828 - acc: 0.9759 - val_loss: 0.
0924 - val_acc: 0.9718
Epoch 3/20
```

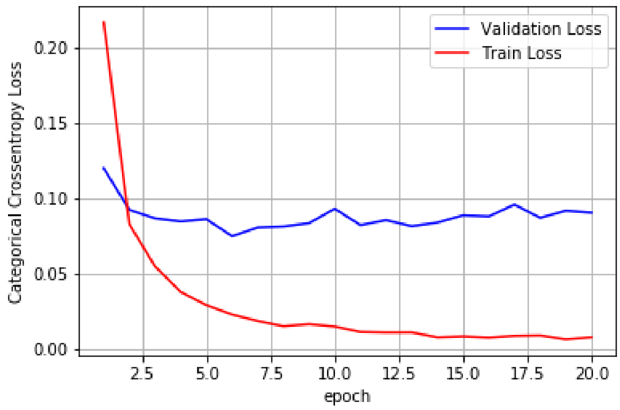
```
60000/60000 [==============================] - 5s 87us/step - loss: 0.0549 - acc: 0.9833 - val_loss: 0.
0867 - val_acc: 0.9731
Epoch 4/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0380 - acc: 0.9889 - val_loss: 0.
0847 - val_acc: 0.9739
Epoch 5/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0293 - acc: 0.9910 - val_loss: 0.
0862 - val_acc: 0.9750
Epoch 6/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0230 - acc: 0.9930 - val_loss: 0.
0749 - val_acc: 0.9774
Epoch 7/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0188 - acc: 0.9942 - val_loss: 0.
0807 - val_acc: 0.9763
Epoch 8/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0153 - acc: 0.9954 - val_loss: 0.
0813 - val_acc: 0.9789
Epoch 9/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0167 - acc: 0.9945 - val_loss: 0.
0835 - val_acc: 0.9756
Epoch 10/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0151 - acc: 0.9954 - val_loss: 0.
0931 - val_acc: 0.9735
Epoch 11/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0115 - acc: 0.9961 - val_loss: 0.
0823 - val_acc: 0.9774
Epoch 12/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0112 - acc: 0.9964 - val_loss: 0.
0856 - val_acc: 0.9769
Epoch 13/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.0112 - acc: 0.9965 - val_loss: 0.
0815 - val_acc: 0.9785
Epoch 14/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.0079 - acc: 0.9974 - val_loss: 0.
0840 - val_acc: 0.9782
Epoch 15/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0085 - acc: 0.9973 - val_loss: 0.
0887 - val_acc: 0.9786
Epoch 16/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.0078 - acc: 0.9977 - val_loss: 0.
0880 - val_acc: 0.9780
Epoch 17/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0089 - acc: 0.9973 - val_loss: 0.
0959 - val_acc: 0.9759
Epoch 18/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0092 - acc: 0.9971 - val_loss: 0.
0871 - val_acc: 0.9774
Epoch 19/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.0066 - acc: 0.9977 - val_loss: 0.
0918 - val_acc: 0.9783
Epoch 20/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.0080 - acc: 0.9974 - val_loss: 0.
0905 - val_acc: 0.9767
```

In [16]:

```
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, validat
ion_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
```

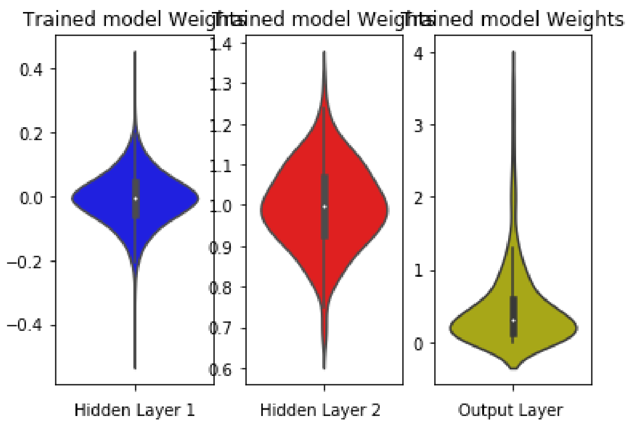
```
# acc : train accuracy
# for each key in histrory.histrory we will have a list of length equal to number of epochs

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

```
Test score: 0.09051545914988383
Test accuracy: 0.9767
```



In [17]:

```
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 [18]:

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

model_drop.add(Dense(72, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed
=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:3733
: 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`.
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_5 (Dense)              (None, 364)               285740    
_________________________________________________________________
batch_normalization_3 (Batch (None, 364)               1456      
_________________________________________________________________
dropout_1 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_6 (Dense)              (None, 72)                26280     
_________________________________________________________________
batch_normalization_4 (Batch (None, 72)                288       
_________________________________________________________________
dropout_2 (Dropout)          (None, 72)                0         
_________________________________________________________________
dense_7 (Dense)              (None, 10)                730       
=================================================================
Total params: 314,494
Trainable params: 313,622
Non-trainable params: 872
_________________________________________________________________
```

In [19]:

```
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, validatio
n_data=(X_test, Y_test))
```

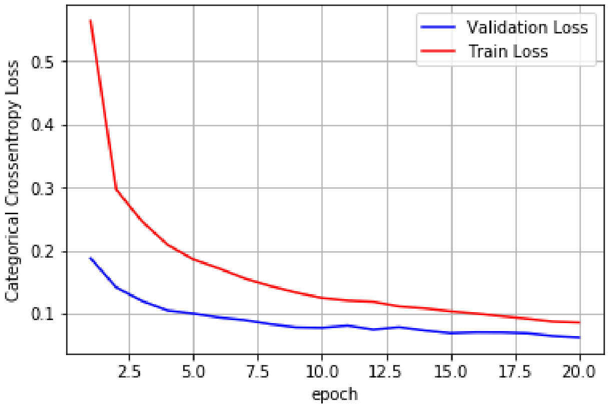
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.5634 - acc: 0.8287 - val_loss: 0
.1879 - val_acc: 0.9407
Epoch 2/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.2963 - acc: 0.9125 - val_loss: 0.
1416 - val_acc: 0.9558
Epoch 3/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.2462 - acc: 0.9276 - val_loss: 0.
1202 - val_acc: 0.9638
Epoch 4/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.2090 - acc: 0.9383 - val_loss: 0.
1052 - val_acc: 0.9660
Epoch 5/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1861 - acc: 0.9445 - val_loss: 0.
1003 - val_acc: 0.9688
```

```
Epoch 6/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.1717 - acc: 0.9499 - val_loss: 0.
0941 - val_acc: 0.9720
Epoch 7/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.1558 - acc: 0.9544 - val_loss: 0.
0899 - val_acc: 0.9725
Epoch 8/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.1437 - acc: 0.9575 - val_loss: 0.
0835 - val_acc: 0.9728
Epoch 9/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.1335 - acc: 0.9595 - val_loss: 0.
0785 - val_acc: 0.9769
Epoch 10/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.1249 - acc: 0.9628 - val_loss: 0.
0778 - val_acc: 0.9764
Epoch 11/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.1209 - acc: 0.9645 - val_loss: 0.
0813 - val_acc: 0.9773
Epoch 12/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.1189 - acc: 0.9647 - val_loss: 0.
0749 - val_acc: 0.9774
Epoch 13/20
60000/60000 [==============================] - 5s 81us/step - loss: 0.1117 - acc: 0.9665 - val_loss: 0.
0788 - val_acc: 0.9774
Epoch 14/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1086 - acc: 0.9674 - val_loss: 0.
0734 - val_acc: 0.9792
Epoch 15/20
60000/60000 [==============================] - 5s 84us/step - loss: 0.1040 - acc: 0.9686 - val_loss: 0.
0693 - val_acc: 0.9802
Epoch 16/20
60000/60000 [==============================] - 5s 82us/step - loss: 0.1003 - acc: 0.9696 - val_loss: 0.
0706 - val_acc: 0.9795
Epoch 17/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0961 - acc: 0.9708 - val_loss: 0.
0704 - val_acc: 0.9810
Epoch 18/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0921 - acc: 0.9727 - val_loss: 0.
0691 - val_acc: 0.9816
Epoch 19/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0876 - acc: 0.9735 - val_loss: 0.
0647 - val_acc: 0.9806
Epoch 20/20
60000/60000 [==============================] - 5s 83us/step - loss: 0.0861 - acc: 0.9730 - val_loss: 0.
0624 - val_acc: 0.9818
```

In [20]:

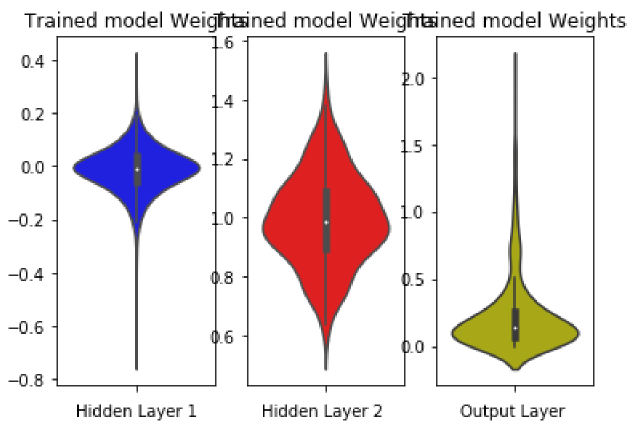
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, validat
ion_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: 0.0624060621908051
Test accuracy: 0.9818
```



In [21]:

```
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()
```



3 Layer MLP+ReLU+Adam

In [22]:

```
model_relu = Sequential()
model_relu.add(Dense(250, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(
mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(100, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, se
```

```python
ed=None)) )
model_relu.add(Dense(85, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, see
d=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, validatio
n_data=(X_test, Y_test))
```

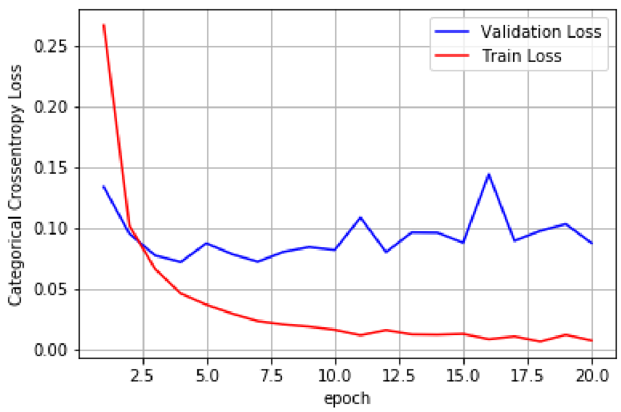
```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_8 (Dense)              (None, 250)               196250    
_________________________________________________________________
dense_9 (Dense)              (None, 100)               25100     
_________________________________________________________________
dense_10 (Dense)             (None, 85)                8585      
_________________________________________________________________
dense_11 (Dense)             (None, 10)                860       
=================================================================
Total params: 230,795
Trainable params: 230,795
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.2668 - acc: 0.9218 - val_loss: 0.
1340 - val_acc: 0.9586
Epoch 2/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.1021 - acc: 0.9693 - val_loss: 0.
0955 - val_acc: 0.9698
Epoch 3/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0665 - acc: 0.9791 - val_loss: 0.
0776 - val_acc: 0.9765
Epoch 4/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.0463 - acc: 0.9858 - val_loss: 0.
0721 - val_acc: 0.9773
Epoch 5/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0371 - acc: 0.9882 - val_loss: 0.
0875 - val_acc: 0.9743
Epoch 6/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0297 - acc: 0.9901 - val_loss: 0.
0786 - val_acc: 0.9767
Epoch 7/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0236 - acc: 0.9923 - val_loss: 0.
0724 - val_acc: 0.9791
Epoch 8/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.0208 - acc: 0.9931 - val_loss: 0.
0805 - val_acc: 0.9791
Epoch 9/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.0190 - acc: 0.9938 - val_loss: 0.
0845 - val_acc: 0.9782
Epoch 10/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.0164 - acc: 0.9944 - val_loss: 0.
0820 - val_acc: 0.9798
Epoch 11/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0119 - acc: 0.9960 - val_loss: 0.
1087 - val_acc: 0.9746
Epoch 12/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0162 - acc: 0.9948 - val_loss: 0.
0803 - val_acc: 0.9818
Epoch 13/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0126 - acc: 0.9961 - val_loss: 0.
0965 - val_acc: 0.9790
Epoch 14/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0124 - acc: 0.9960 - val_loss: 0.
0963 - val_acc: 0.9781
Epoch 15/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0130 - acc: 0.9960 - val_loss: 0.
0880 - val_acc: 0.9798
Epoch 16/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.0086 - acc: 0.9972 - val_loss: 0.
```

```
60000/60000 [==============================] - 3s 54us/step - loss: 0.0086 - acc: 0.9972 - val_loss: 0.
1445 - val_acc: 0.9678
Epoch 17/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.0109 - acc: 0.9963 - val_loss: 0.
0896 - val_acc: 0.9798
Epoch 18/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0067 - acc: 0.9978 - val_loss: 0.
0978 - val_acc: 0.9794
Epoch 19/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0123 - acc: 0.9960 - val_loss: 0.
1034 - val_acc: 0.9791
Epoch 20/20
60000/60000 [==============================] - 3s 53us/step - loss: 0.0076 - acc: 0.9978 - val_loss: 0.
0878 - val_acc: 0.9827
```

In [23]:

```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(x)

# 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, validat
ion_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']
#print(ty)
plt_dynamic(x, vy, ty, ax)
```

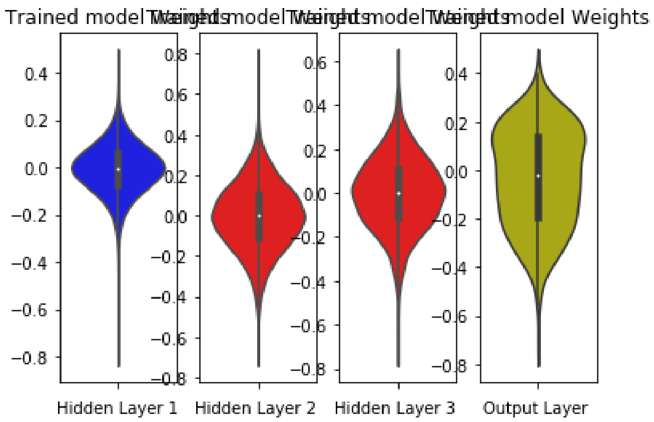
```
Test score: 0.087810692615682
Test accuracy: 0.9827
```



In [26]:

```python
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)
```

```
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='r')
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()
```



3 Layer MLP+Batch Normalization+ReLU+Adam

In [27]:

```
# 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(250, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal
(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(100, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, se
ed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(85, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, see
d=None)) )
model_batch.add(BatchNormalization())

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

```
model_batch.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_12 (Dense)             (None, 250)               196250    
_________________________________________________________________
batch_normalization_5 (Batch (None, 250)               1000      
_________________________________________________________________
dense_13 (Dense)             (None, 100)               25100     
_________________________________________________________________
batch_normalization_6 (Batch (None, 100)               400       
_________________________________________________________________
dense_14 (Dense)             (None, 85)                8585      
_________________________________________________________________
batch_normalization_7 (Batch (None, 85)                340       
_________________________________________________________________
dense_15 (Dense)             (None, 10)                860       
=================================================================
Total params: 232,535
Trainable params: 231,665
Non-trainable params: 870
_________________________________________________________________
```

In [28]:

```
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, validati
on_data=(X_test, Y_test))
```

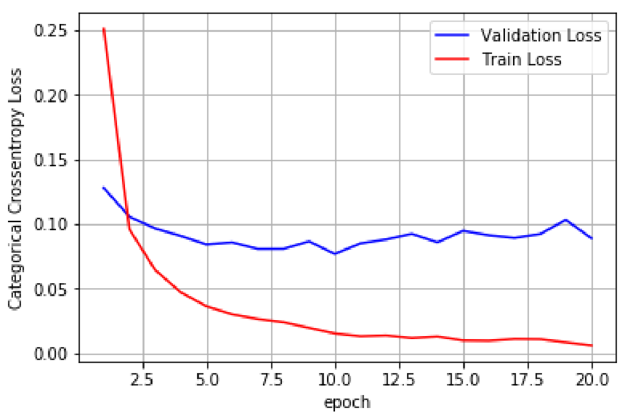
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.2506 - acc: 0.9276 - val_loss: 0
.1278 - val_acc: 0.9606
Epoch 2/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0959 - acc: 0.9715 - val_loss: 0.
1056 - val_acc: 0.9669
Epoch 3/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0645 - acc: 0.9806 - val_loss: 0.
0965 - val_acc: 0.9693
Epoch 4/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0471 - acc: 0.9851 - val_loss: 0
.0907 - val_acc: 0.9718
Epoch 5/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0364 - acc: 0.9891 - val_loss: 0.
0841 - val_acc: 0.9746
Epoch 6/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0302 - acc: 0.9905 - val_loss: 0.
0857 - val_acc: 0.9755
Epoch 7/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0265 - acc: 0.9912 - val_loss: 0
.0807 - val_acc: 0.9755
Epoch 8/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0240 - acc: 0.9923 - val_loss: 0.
0807 - val_acc: 0.9759
Epoch 9/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0197 - acc: 0.9935 - val_loss: 0.
0865 - val_acc: 0.9750
Epoch 10/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.0153 - acc: 0.9952 - val_loss: 0.
0768 - val_acc: 0.9784
Epoch 11/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0132 - acc: 0.9959 - val_loss: 0.
0849 - val_acc: 0.9762
Epoch 12/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.0138 - acc: 0.9952 - val_loss: 0.
0880 - val_acc: 0.9776
Epoch 13/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.0118 - acc: 0.9960 - val_loss: 0.
0923 - val_acc: 0.9762
Epoch 14/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0130 - acc: 0.9956 - val_loss: 0
```

```
.0858 - val_acc: 0.9783
Epoch 15/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0100 - acc: 0.9967 - val_loss: 0.
0947 - val_acc: 0.9774
Epoch 16/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0097 - acc: 0.9967 - val_loss: 0.
0912 - val_acc: 0.9781
Epoch 17/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0111 - acc: 0.9963 - val_loss: 0
.0892 - val_acc: 0.9783
Epoch 18/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0109 - acc: 0.9965 - val_loss: 0
.0922 - val_acc: 0.9764
Epoch 19/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0085 - acc: 0.9971 - val_loss: 0.
1031 - val_acc: 0.9772
Epoch 20/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0061 - acc: 0.9982 - val_loss: 0
.0889 - val_acc: 0.9800
```

In [29]:

```
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, validat
ion_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)
```

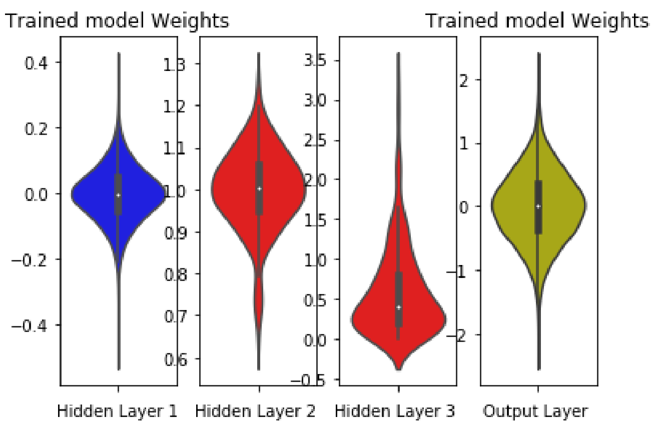
```
Test score: 0.08889246554333222
Test accuracy: 0.98
```



In [30]:

```
w_after = model_batch.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
```

```
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='r')
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()
```



3 Layer MLP+ReLU+Dropout+Adam

In [31]:

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

model_drop.add(Dense(100, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, see
d=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(85, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed
=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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

```
model_drop.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_16 (Dense)             (None, 250)               196250    
_________________________________________________________________
batch_normalization_8 (Batch (None, 250)               1000      
_________________________________________________________________
dropout_3 (Dropout)          (None, 250)               0         
_________________________________________________________________
dense_17 (Dense)             (None, 100)               25100     
_________________________________________________________________
batch_normalization_9 (Batch (None, 100)               400       
_________________________________________________________________
dropout_4 (Dropout)          (None, 100)               0         
_________________________________________________________________
dense_18 (Dense)             (None, 85)                8585      
_________________________________________________________________
batch_normalization_10 (Batc (None, 85)                340       
_________________________________________________________________
dropout_5 (Dropout)          (None, 85)                0         
_________________________________________________________________
dense_19 (Dense)             (None, 10)                860       
=================================================================
Total params: 232,535
Trainable params: 231,665
Non-trainable params: 870
_________________________________________________________________
```

In [32]:

```
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, validatio
n_data=(X_test, Y_test))
```

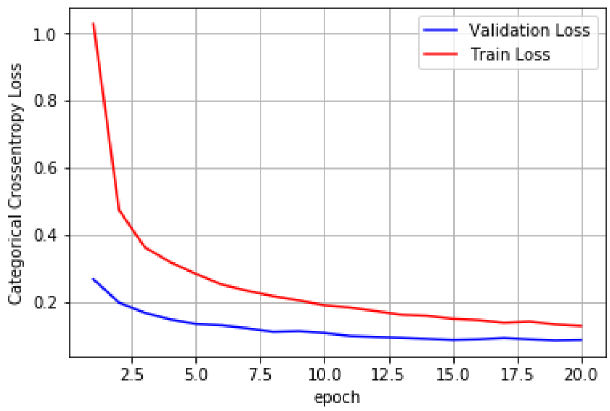
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 128us/step - loss: 1.0290 - acc: 0.6731 - val_loss: 0
.2667 - val_acc: 0.9234
Epoch 2/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.4735 - acc: 0.8611 - val_loss: 0
.1968 - val_acc: 0.9387
Epoch 3/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.3619 - acc: 0.8965 - val_loss: 0
.1666 - val_acc: 0.9508
Epoch 4/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.3176 - acc: 0.9091 - val_loss: 0
.1467 - val_acc: 0.9554
Epoch 5/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.2827 - acc: 0.9201 - val_loss: 0
.1331 - val_acc: 0.9597
Epoch 6/20
60000/60000 [==============================] - 6s 107us/step - loss: 0.2512 - acc: 0.9293 - val_loss: 0
.1291 - val_acc: 0.9605
Epoch 7/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.2323 - acc: 0.9352 - val_loss: 0
.1202 - val_acc: 0.9652
Epoch 8/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.2164 - acc: 0.9398 - val_loss: 0
.1096 - val_acc: 0.9664
Epoch 9/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.2037 - acc: 0.9433 - val_loss: 0
.1117 - val_acc: 0.9674
Epoch 10/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.1891 - acc: 0.9478 - val_loss: 0
.1063 - val_acc: 0.9697
Epoch 11/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.1822 - acc: 0.9496 - val_loss: 0
.0971 - val_acc: 0.9712
Epoch 12/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.1717 - acc: 0.9513 - val_loss: 0
.0944 - val_acc: 0.9734
```

```
Epoch 13/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.1608 - acc: 0.9560 - val_loss: 0
.0919 - val_acc: 0.9739
Epoch 14/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.1577 - acc: 0.9553 - val_loss: 0
.0886 - val_acc: 0.9754
Epoch 15/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.1487 - acc: 0.9583 - val_loss: 0
.0853 - val_acc: 0.9754
Epoch 16/20
60000/60000 [==============================] - 6s 106us/step - loss: 0.1448 - acc: 0.9597 - val_loss: 0
.0872 - val_acc: 0.9758
Epoch 17/20
60000/60000 [==============================] - 6s 103us/step - loss: 0.1369 - acc: 0.9622 - val_loss: 0
.0914 - val_acc: 0.9757
Epoch 18/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.1405 - acc: 0.9615 - val_loss: 0
.0871 - val_acc: 0.9762
Epoch 19/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.1317 - acc: 0.9621 - val_loss: 0
.0839 - val_acc: 0.9758
Epoch 20/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.1272 - acc: 0.9645 - val_loss: 0
.0853 - val_acc: 0.9760
```

In [33]:

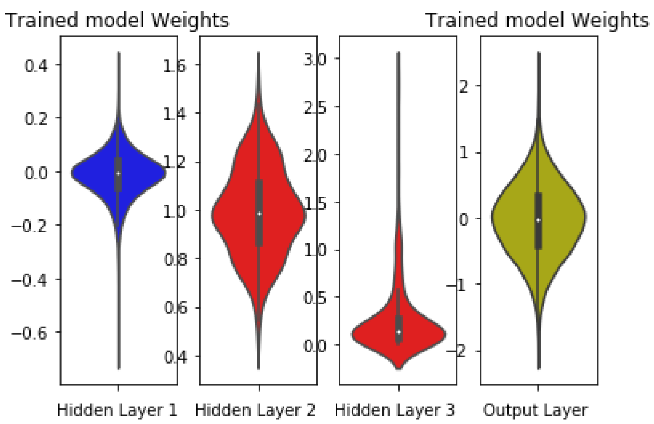
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, validat
ion_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: 0.08526910331975669
Test accuracy: 0.976
```



In [34]:

```
w_after = model_drop.get_weights()

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


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

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

plt.subplot(1, 4, 3)
#plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='r')
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()
```



5 Layer MLP+ReLU+Adam

In [35]:

```
model_relu = Sequential()
model_relu.add(Dense(235, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(
mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(102, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, se
ed=None)) )
model_relu.add(Dense(65, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, see
d=None)) )
model_relu.add(Dense(50, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, see
d=None)) )
model_relu.add(Dense(35, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, see
d=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, validatio
n_data=(X_test, Y_test))
```

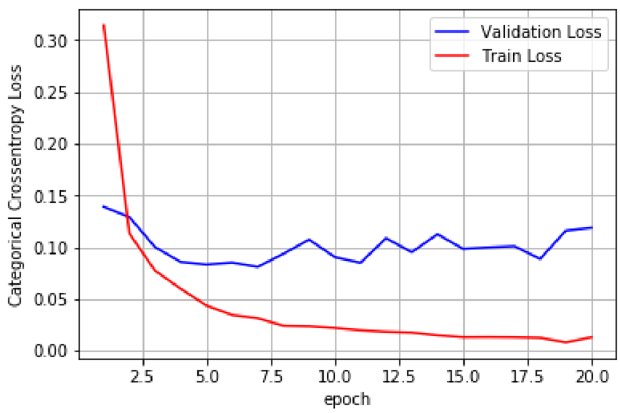
```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_20 (Dense)             (None, 235)               184475
_________________________________________________________________
dense_21 (Dense)             (None, 102)               24072
_________________________________________________________________
dense_22 (Dense)             (None, 65)                6695
_________________________________________________________________
dense_23 (Dense)             (None, 50)                3300
_________________________________________________________________
dense_24 (Dense)             (None, 35)                1785
_________________________________________________________________
dense_25 (Dense)             (None, 10)                360
=================================================================
Total params: 220,687
Trainable params: 220,687
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 79us/step - loss: 0.3140 - acc: 0.9043 - val_loss: 0.
1389 - val_acc: 0.9569
Epoch 2/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.1132 - acc: 0.9657 - val_loss: 0.
1291 - val_acc: 0.9598
Epoch 3/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0774 - acc: 0.9762 - val_loss: 0.
1000 - val_acc: 0.9690
Epoch 4/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0598 - acc: 0.9810 - val_loss: 0.
0857 - val_acc: 0.9748
Epoch 5/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.0437 - acc: 0.9863 - val_loss: 0.
0832 - val_acc: 0.9769
Epoch 6/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0346 - acc: 0.9893 - val_loss: 0.
0851 - val_acc: 0.9767
Epoch 7/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0315 - acc: 0.9896 - val_loss: 0.
0810 - val_acc: 0.9772
Epoch 8/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.0242 - acc: 0.9918 - val_loss: 0.
0936 - val_acc: 0.9745
Epoch 9/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0237 - acc: 0.9920 - val_loss: 0.
1073 - val_acc: 0.9735
Epoch 10/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0221 - acc: 0.9927 - val_loss: 0.
0905 - val_acc: 0.9774
Epoch 11/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0198 - acc: 0.9936 - val_loss: 0.
0849 - val_acc: 0.9778
Epoch 12/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0183 - acc: 0.9938 - val_loss: 0.
1086 - val_acc: 0.9759
Epoch 13/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0176 - acc: 0.9945 - val_loss: 0.
0952 - val_acc: 0.9766
Epoch 14/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.0149 - acc: 0.9951 - val_loss: 0.
1125 - val_acc: 0.9751
Epoch 15/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0131 - acc: 0.9959 - val_loss: 0.
0983 - val_acc: 0.9779
Epoch 16/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0132 - acc: 0.9959 - val_loss: 0.
0997 - val_acc: 0.9783
Epoch 17/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0130 - acc: 0.9959 - val_loss: 0.
1009 - val_acc: 0.9797
Epoch 18/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0125 - acc: 0.9964 - val_loss: 0.
0887 - val_acc: 0.9814
```

```
Epoch 19/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0081 - acc: 0.9975 - val_loss: 0.
1159 - val_acc: 0.9765
Epoch 20/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0130 - acc: 0.9959 - val_loss: 0.
1187 - val_acc: 0.9765
```

In [36]:

```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, validat
ion_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)
```

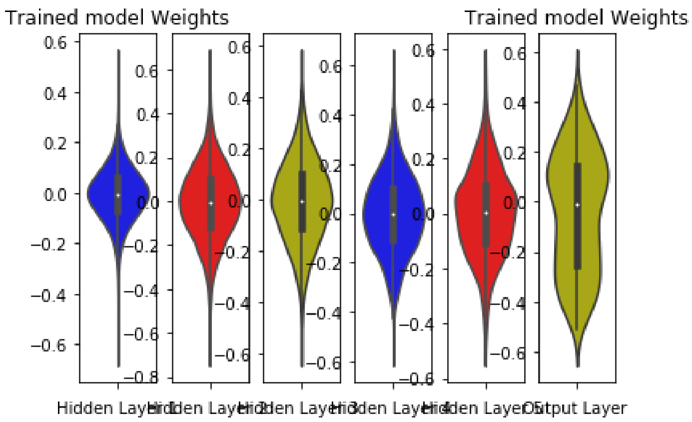
```
Test score: 0.11873800104274924
Test accuracy: 0.9765
```



In [38]:

```python
w_after = model_relu.get_weights()

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


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

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

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

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

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

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

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



5 Layer MLP+ Batch normalization_Relu_Adam

In [39]:

```
# 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(235, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal
(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(102, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, se
ed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(65, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, see
d=None)) )
model_batch.add(BatchNormalization())
```

```
model_batch.add(Dense(50, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, see
d=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(35, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, see
d=None)) )
model_batch.add(BatchNormalization())


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


model_batch.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_26 (Dense)             (None, 235)               184475
_________________________________________________________________
batch_normalization_11 (Batc (None, 235)               940
_________________________________________________________________
dense_27 (Dense)             (None, 102)               24072
_________________________________________________________________
batch_normalization_12 (Batc (None, 102)               408
_________________________________________________________________
dense_28 (Dense)             (None, 65)                6695
_________________________________________________________________
batch_normalization_13 (Batc (None, 65)                260
_________________________________________________________________
dense_29 (Dense)             (None, 50)                3300
_________________________________________________________________
batch_normalization_14 (Batc (None, 50)                200
_________________________________________________________________
dense_30 (Dense)             (None, 35)                1785
_________________________________________________________________
batch_normalization_15 (Batc (None, 35)                140
_________________________________________________________________
dense_31 (Dense)             (None, 10)                360
=================================================================
Total params: 222,635
Trainable params: 221,661
Non-trainable params: 974
_________________________________________________________________
```

In [40]:

```
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, validati
on_data=(X_test, Y_test))
```

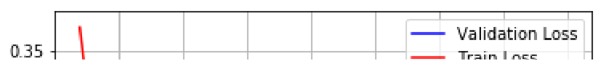
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 11s 183us/step - loss: 0.3773 - acc: 0.8937 - val_loss:
0.1553 - val_acc: 0.9558
Epoch 2/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.1254 - acc: 0.9630 - val_loss: 0
.1210 - val_acc: 0.9621
Epoch 3/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.0879 - acc: 0.9733 - val_loss: 0
.1143 - val_acc: 0.9656
Epoch 4/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.0680 - acc: 0.9791 - val_loss: 0
.1035 - val_acc: 0.9705
Epoch 5/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.0527 - acc: 0.9833 - val_loss: 0
.1121 - val_acc: 0.9672
Epoch 6/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0434 - acc: 0.9863 - val_loss: 0
.1037 - val_acc: 0.9702
Epoch 7/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0364 - acc: 0.9882 - val_loss: 0
.1156 - val_acc: 0.9676
```

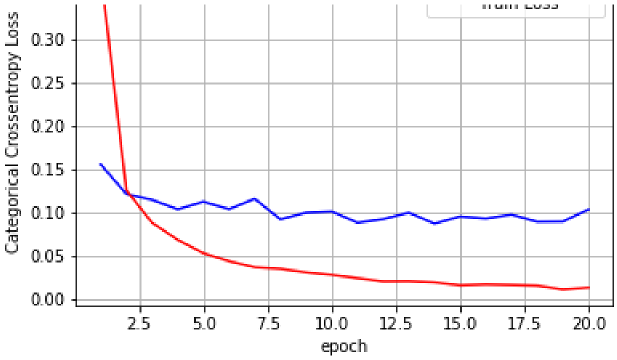
```
Epoch 8/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.0346 - acc: 0.9883 - val_loss: 0
.0918 - val_acc: 0.9739
Epoch 9/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0305 - acc: 0.9895 - val_loss: 0
.0995 - val_acc: 0.9735
Epoch 10/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0276 - acc: 0.9914 - val_loss: 0
.1009 - val_acc: 0.9726
Epoch 11/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0238 - acc: 0.9922 - val_loss: 0
.0880 - val_acc: 0.9767
Epoch 12/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.0200 - acc: 0.9933 - val_loss: 0
.0920 - val_acc: 0.9772
Epoch 13/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0201 - acc: 0.9935 - val_loss: 0
.0994 - val_acc: 0.9717
Epoch 14/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.0189 - acc: 0.9934 - val_loss: 0
.0869 - val_acc: 0.9772
Epoch 15/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0155 - acc: 0.9952 - val_loss: 0
.0950 - val_acc: 0.9758
Epoch 16/20
60000/60000 [==============================] - 9s 145us/step - loss: 0.0164 - acc: 0.9945 - val_loss: 0
.0927 - val_acc: 0.9765
Epoch 17/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0158 - acc: 0.9946 - val_loss: 0
.0972 - val_acc: 0.9758
Epoch 18/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.0151 - acc: 0.9948 - val_loss: 0
.0891 - val_acc: 0.9789
Epoch 19/20
60000/60000 [==============================] - 9s 144us/step - loss: 0.0108 - acc: 0.9964 - val_loss: 0
.0893 - val_acc: 0.9773
Epoch 20/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.0129 - acc: 0.9956 - val_loss: 0
.1032 - val_acc: 0.9754
```

In [41]:

```
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, validat
ion_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)
```

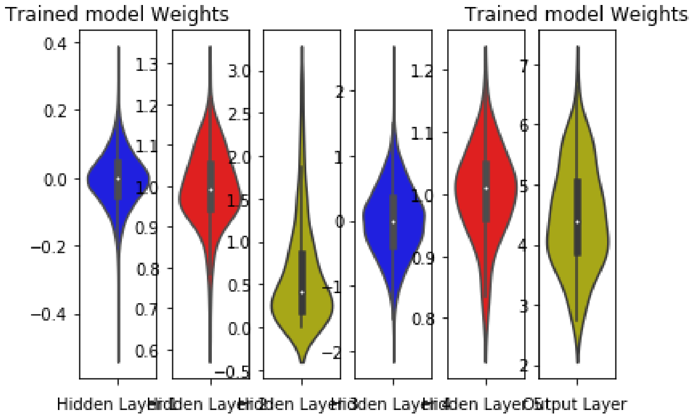
```
Test score: 0.10316739412156312
Test accuracy: 0.9754
```





In [42]:

```
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)


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

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

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

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

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

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



5 MLP+Dropout+ReLU+Adam

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

model_drop.add(Dense(102, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, see
d=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(65, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed
=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(50, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed
=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(35, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed
=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))


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


model_drop.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_32 (Dense)             (None, 235)               184475    
_________________________________________________________________
batch_normalization_16 (Batc (None, 235)               940       
_________________________________________________________________
dropout_6 (Dropout)          (None, 235)               0         
_________________________________________________________________
dense_33 (Dense)             (None, 102)               24072     
_________________________________________________________________
batch_normalization_17 (Batc (None, 102)               408       
_________________________________________________________________
dropout_7 (Dropout)          (None, 102)               0         
_________________________________________________________________
dense_34 (Dense)             (None, 65)                6695      
_________________________________________________________________
batch_normalization_18 (Batc (None, 65)                260       
_________________________________________________________________
dropout_8 (Dropout)          (None, 65)                0         
_________________________________________________________________
dense_35 (Dense)             (None, 50)                3300      
_________________________________________________________________
batch_normalization_19 (Batc (None, 50)                200       
_________________________________________________________________
dropout_9 (Dropout)          (None, 50)                0         
_________________________________________________________________
dense_36 (Dense)             (None, 35)                1785      
_________________________________________________________________
batch_normalization_20 (Batc (None, 35)                140       
```

```
_________________________________________________________________
dropout_10 (Dropout)         (None, 35)                0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                360
=================================================================
Total params: 222,635
Trainable params: 221,661
Non-trainable params: 974
_________________________________________________________________
```

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, validatio
n_data=(X_test, Y_test))
```

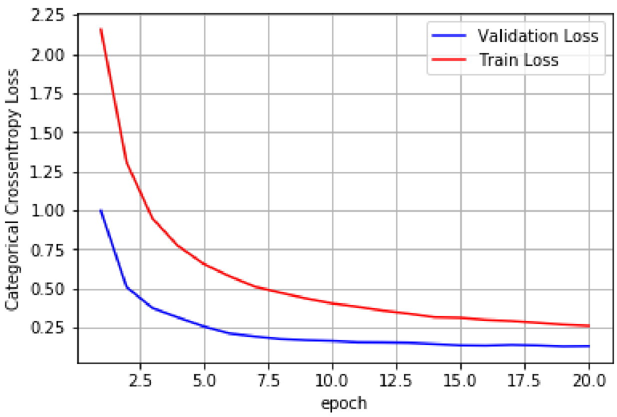
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 12s 199us/step - loss: 2.1567 - acc: 0.2681 - val_loss:
0.9967 - val_acc: 0.7442
Epoch 2/20
60000/60000 [==============================] - 9s 152us/step - loss: 1.3065 - acc: 0.5476 - val_loss: 0
.5086 - val_acc: 0.8687
Epoch 3/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.9505 - acc: 0.6814 - val_loss: 0
.3757 - val_acc: 0.8998
Epoch 4/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.7723 - acc: 0.7492 - val_loss: 0
.3148 - val_acc: 0.9044
Epoch 5/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.6576 - acc: 0.7985 - val_loss: 0
.2575 - val_acc: 0.9332
Epoch 6/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.5789 - acc: 0.8294 - val_loss: 0
.2122 - val_acc: 0.9481
Epoch 7/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.5112 - acc: 0.8555 - val_loss: 0
.1928 - val_acc: 0.9506
Epoch 8/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.4727 - acc: 0.8710 - val_loss: 0
.1762 - val_acc: 0.9554
Epoch 9/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.4358 - acc: 0.8865 - val_loss: 0
.1687 - val_acc: 0.9554
Epoch 10/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.4043 - acc: 0.8961 - val_loss: 0
.1642 - val_acc: 0.9589
Epoch 11/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.3823 - acc: 0.9035 - val_loss: 0
.1555 - val_acc: 0.9621
Epoch 12/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.3582 - acc: 0.9100 - val_loss: 0
.1546 - val_acc: 0.9620
Epoch 13/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.3387 - acc: 0.9183 - val_loss: 0
.1523 - val_acc: 0.9631
Epoch 14/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.3162 - acc: 0.9233 - val_loss: 0
.1443 - val_acc: 0.9649
Epoch 15/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.3127 - acc: 0.9246 - val_loss: 0
.1361 - val_acc: 0.9681
Epoch 16/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.2990 - acc: 0.9295 - val_loss: 0
.1344 - val_acc: 0.9675
Epoch 17/20
60000/60000 [==============================] - 10s 161us/step - loss: 0.2919 - acc: 0.9312 - val_loss:
0.1393 - val_acc: 0.9681
Epoch 18/20
60000/60000 [==============================] - 9s 153us/step - loss: 0.2806 - acc: 0.9355 - val_loss: 0
.1357 - val_acc: 0.9678
Epoch 19/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.2688 - acc: 0.9363 - val_loss: 0
.1293 - val_acc: 0.9715
Epoch 20/20
```

```
Epoch 20/20
60000/60000 [==============================] - 9s 152us/step - loss: 0.2613 - acc: 0.9396 - val_loss: 0
.1308 - val_acc: 0.9706
```

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, validat
ion_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)
```

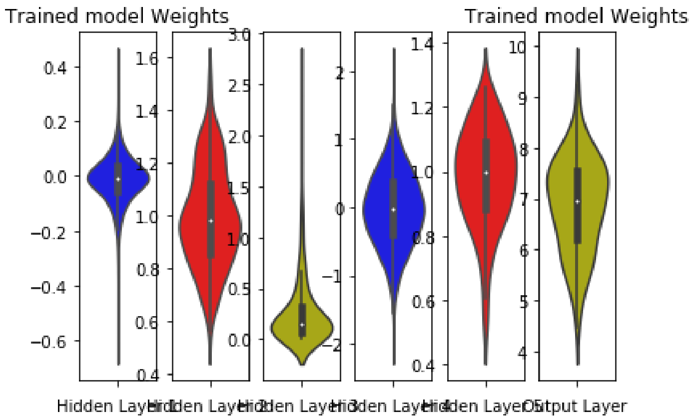
```
Test score: 0.13083188042640687
Test accuracy: 0.9706
```



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


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

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

```
plt.xlabel('Hidden Layer 2 ')

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

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

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

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



In [49]:

```
from prettytable import PrettyTable
x = PrettyTable(["Feature", "Accuracy(%)", "Loss(%)"])
x.add_row(["ReLU+Adam", 98.1, 9.2])
x.add_row(["ReLU+Batch+Adam",97.6,9.05])
x.add_row(["ReLU+Dropout+Adam",98.18,6.24])
print("2 Layer MLP Results")
print(x)


y = PrettyTable(["Feature", "Accuracy(%)", "Loss(%)"])
y.add_row(["ReLU+Adam",98.27, 8.78])
y.add_row(["ReLU+Batch+Adam",98,8.88])
y.add_row(["ReLU+Dropout+Adam",97.6,8.52])
print("3 Layer MLP Results")
print(y)



z = PrettyTable(["Feature", "Accuracy(%)", "Loss(%)"])
z.add_row(["ReLU+Adam", 97.65, 11.87])
z.add_row(["ReLU+Batch+Adam",97.54,10.31])
z.add_row(["ReLU+Dropout+Adam",97.06,13.08])
print("5 Layer MLP Results")
print(z)
```

```
2 Layer MLP Results
+-------------------+-------------+---------+
|      Feature      | Accuracy(%) | Loss(%) |
+-------------------+-------------+---------+
```

ReLU+Adam	98.1	9.2
ReLU+Batch+Adam	97.6	9.05
ReLU+Dropout+Adam	98.18	6.24

3 Layer MLP Results

Feature	Accuracy(%)	Loss(%)
ReLU+Adam	98.27	8.78
ReLU+Batch+Adam	98	8.88
ReLU+Dropout+Adam	97.6	8.52

5 Layer MLP Results

Feature	Accuracy(%)	Loss(%)
ReLU+Adam	97.65	11.87
ReLU+Batch+Adam	97.54	10.31
ReLU+Dropout+Adam	97.06	13.08

Observation:

1. For 2 hidden layers, the number of nodes taken in each layer are 364 and 72.
2. For 3 hidden layers, the number of nodes taken in each layer are 250,100,85.
3. For 5 hidden layers, the number of nodes taken in each layer are 235,102,65,50,35.
4. ReLU activation function with Adam optimizer is implemented with Batch normalization and dropout.
5. Upon seeing the results, for 3 MLP the Multi-class log loss is lesser(8.52) compared to 5 MLP(>10).
6. With respect to accuracy, again 3 MLP produces better results(98.27).