Keras -- MLPs on MNIST

In [1]:

```
%tensorflow_version 1.9
# if your keras is not using tensorflow as backend set "KERAS_BACKEND=tensorflow" use t
his command
from keras.utils import np_utils
from keras.datasets import mnist
import seaborn as sns
from keras.initializers import RandomNormal
```

```
`%tensorflow_version` only switches the major version: 1.x or 2.x.
You set: `1.9`. This will be interpreted as: `1.x`.


TensorFlow 1.x selected.

Using TensorFlow backend.
```

In [0]:

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
import time
%matplotlib inline
# 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 [0]:

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

In [4]:

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

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

In [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 [6]:

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

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

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

In [7]:

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

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

In [0]:

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

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

In [9]:

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

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

```
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.1372549  0.94509804
 0.88235294 0.62745098 0.42352941 0.00392157 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         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 constru
ctor:

# 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_re
gularizer=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 supported by all forward layers:

# from keras.layers import Activation, Dense

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

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

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


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

In [0]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [13]:

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

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

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

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

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

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

In [14]:

```
# Before training a model, you need to configure the learning process, which is done vi
a the compile 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 metric
s=['accuracy'].  https://keras.io/metrics/


# Note: when using the categorical_crossentropy loss, your targets should be in categor
ical format
# (e.g. if you have 10 classes, the target for each sample should be a 10-dimensional v
ector 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, valid
ation_split=0.0,
# validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_ep
och=0, steps_per_epoch=None,
# validation_steps=None)

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

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

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

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

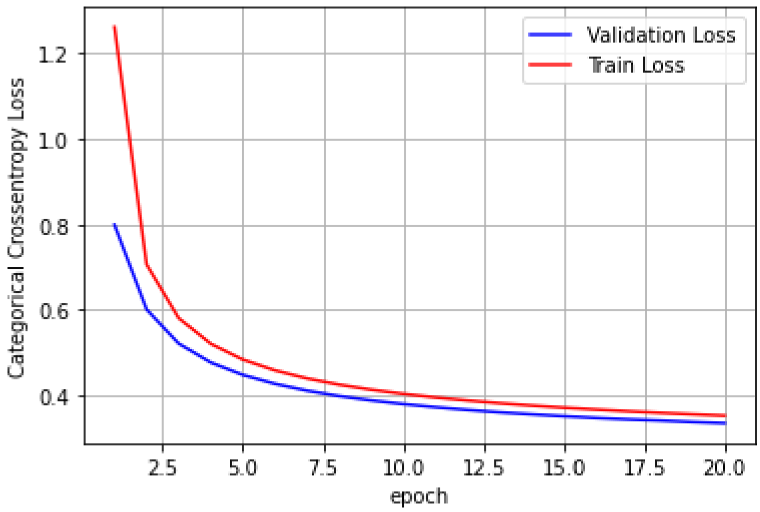
```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/optim
izers.py:793: The name tf.train.Optimizer is deprecated. Please use tf.com
pat.v1.train.Optimizer instead.

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

WARNING:tensorflow:From /tensorflow-1.15.2/python3.6/tensorflow_core/pytho
n/ops/math_grad.py:1424: where (from tensorflow.python.ops.array_ops) is d
eprecated 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
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:1033: The name tf.assign_add is deprecated. Pleas
e use tf.compat.v1.assign_add instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:1020: The name tf.assign is deprecated. Please us
e tf.compat.v1.assign instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:3005: The name tf.Session is deprecated. Please u
se tf.compat.v1.Session instead.

Train on 60000 samples, validate on 10000 samples
Epoch 1/20
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:190: The name tf.get_default_session is deprecate
d. Please use tf.compat.v1.get_default_session instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:197: The name tf.ConfigProto is deprecated. Pleas
e use tf.compat.v1.ConfigProto instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:207: The name tf.global_variables is deprecated.
Please use tf.compat.v1.global_variables instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:216: The name tf.is_variable_initialized is depre
cated. Please use tf.compat.v1.is_variable_initialized instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:223: The name tf.variables_initializer is depreca
ted. Please use tf.compat.v1.variables_initializer instead.

60000/60000 [==============================] - 3s 42us/step - loss: 1.2616
- acc: 0.7113 - val_loss: 0.7992 - val_acc: 0.8424
Epoch 2/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.7065
- acc: 0.8454 - val_loss: 0.6013 - val_acc: 0.8665
Epoch 3/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.5810
- acc: 0.8638 - val_loss: 0.5215 - val_acc: 0.8770
Epoch 4/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.5208
- acc: 0.8727 - val_loss: 0.4771 - val_acc: 0.8828
Epoch 5/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4840
- acc: 0.8782 - val_loss: 0.4482 - val_acc: 0.8873
```

```
Epoch 6/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4588
- acc: 0.8816 - val_loss: 0.4274 - val_acc: 0.8911
Epoch 7/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.4400
- acc: 0.8849 - val_loss: 0.4114 - val_acc: 0.8943
Epoch 8/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.4254
- acc: 0.8870 - val_loss: 0.3987 - val_acc: 0.8957
Epoch 9/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.4137
- acc: 0.8896 - val_loss: 0.3886 - val_acc: 0.8977
Epoch 10/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.4039
- acc: 0.8909 - val_loss: 0.3801 - val_acc: 0.8997
Epoch 11/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.3955
- acc: 0.8927 - val_loss: 0.3728 - val_acc: 0.9007
Epoch 12/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3884
- acc: 0.8943 - val_loss: 0.3664 - val_acc: 0.9019
Epoch 13/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.3821
- acc: 0.8954 - val_loss: 0.3607 - val_acc: 0.9035
Epoch 14/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3766
- acc: 0.8968 - val_loss: 0.3561 - val_acc: 0.9042
Epoch 15/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3717
- acc: 0.8978 - val_loss: 0.3517 - val_acc: 0.9044
Epoch 16/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3672
- acc: 0.8988 - val_loss: 0.3475 - val_acc: 0.9057
Epoch 17/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3632
- acc: 0.8996 - val_loss: 0.3442 - val_acc: 0.9066
Epoch 18/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3595
- acc: 0.9003 - val_loss: 0.3415 - val_acc: 0.9072
Epoch 19/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.3561
- acc: 0.9010 - val_loss: 0.3381 - val_acc: 0.9083
Epoch 20/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.3530
- acc: 0.9020 - val_loss: 0.3354 - val_acc: 0.9087
```

In [15]:

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

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

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

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

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

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

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

```
Test score: 0.3353958689153194
Test accuracy: 0.9087
```



MLP + Sigmoid activation + SGDOptimizer

In [16]:

```
# Multilayer perceptron

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

model_sigmoid.summary()
```

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

In [17]:

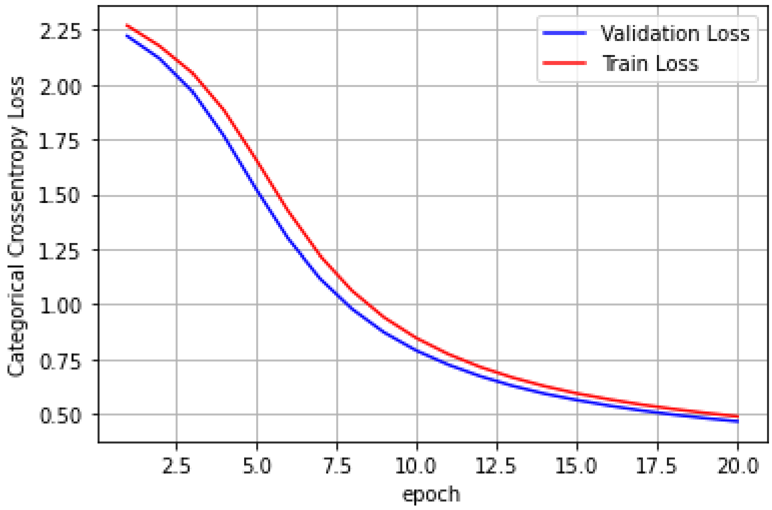
```
model_sigmoid.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accur
acy'])

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, v
erbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 31us/step - loss: 2.2699
- acc: 0.2203 - val_loss: 2.2220 - val_acc: 0.4296
Epoch 2/20
60000/60000 [==============================] - 2s 28us/step - loss: 2.1776
- acc: 0.4451 - val_loss: 2.1204 - val_acc: 0.5472
Epoch 3/20
60000/60000 [==============================] - 2s 27us/step - loss: 2.0578
- acc: 0.5688 - val_loss: 1.9739 - val_acc: 0.6192
Epoch 4/20
60000/60000 [==============================] - 2s 26us/step - loss: 1.8843
- acc: 0.6262 - val_loss: 1.7666 - val_acc: 0.6545
Epoch 5/20
60000/60000 [==============================] - 2s 27us/step - loss: 1.6600
- acc: 0.6701 - val_loss: 1.5255 - val_acc: 0.7121
Epoch 6/20
60000/60000 [==============================] - 2s 27us/step - loss: 1.4250
- acc: 0.7124 - val_loss: 1.2996 - val_acc: 0.7272
Epoch 7/20
60000/60000 [==============================] - 2s 26us/step - loss: 1.2214
- acc: 0.7447 - val_loss: 1.1173 - val_acc: 0.7667
Epoch 8/20
60000/60000 [==============================] - 2s 27us/step - loss: 1.0620
- acc: 0.7711 - val_loss: 0.9789 - val_acc: 0.7865
Epoch 9/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.9404
- acc: 0.7911 - val_loss: 0.8718 - val_acc: 0.8033
Epoch 10/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.8467
- acc: 0.8069 - val_loss: 0.7894 - val_acc: 0.8172
Epoch 11/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.7731
- acc: 0.8193 - val_loss: 0.7247 - val_acc: 0.8274
Epoch 12/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.7141
- acc: 0.8287 - val_loss: 0.6719 - val_acc: 0.8391
Epoch 13/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.6663
- acc: 0.8372 - val_loss: 0.6284 - val_acc: 0.8455
Epoch 14/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.6269
- acc: 0.8447 - val_loss: 0.5925 - val_acc: 0.8504
Epoch 15/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.5940
- acc: 0.8502 - val_loss: 0.5635 - val_acc: 0.8569
Epoch 16/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.5664
- acc: 0.8552 - val_loss: 0.5378 - val_acc: 0.8632
Epoch 17/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.5429
- acc: 0.8597 - val_loss: 0.5158 - val_acc: 0.8664
Epoch 18/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.5225
- acc: 0.8641 - val_loss: 0.4970 - val_acc: 0.8707
Epoch 19/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.5048
- acc: 0.8673 - val_loss: 0.4806 - val_acc: 0.8739
Epoch 20/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.4890
- acc: 0.8700 - val_loss: 0.4663 - val_acc: 0.8771
```

In [18]:

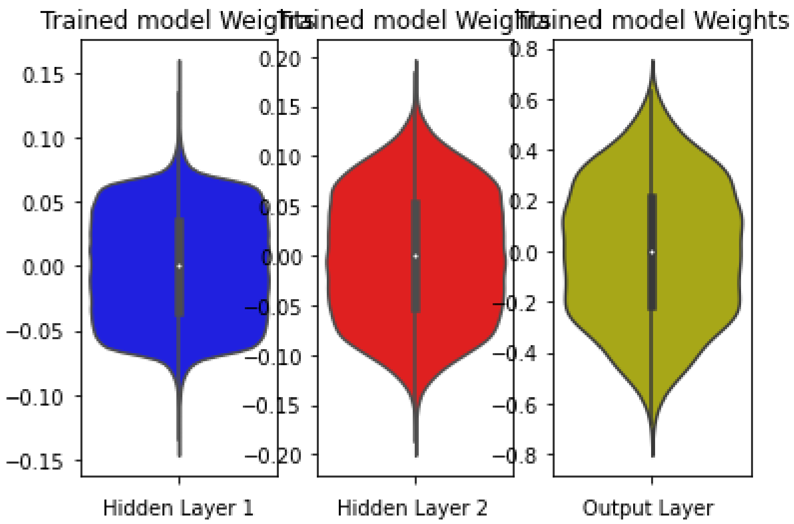
```
score = model_sigmoid.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.46628043448925016
Test accuracy: 0.8771
```



In [19]:

```
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 [20]:

```
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=['accu
racy'])

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, v
erbose=1, validation_data=(X_test, Y_test))
```

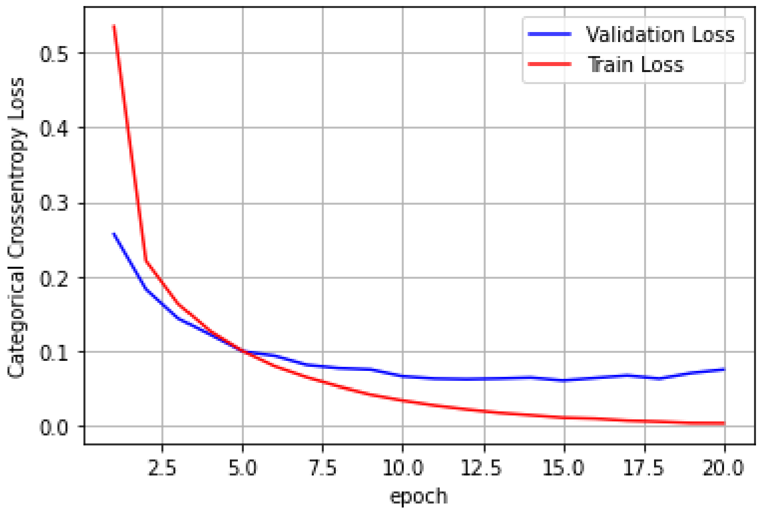
```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_6 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.5341
- acc: 0.8603 - val_loss: 0.2566 - val_acc: 0.9235
Epoch 2/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.2208
- acc: 0.9350 - val_loss: 0.1828 - val_acc: 0.9462
Epoch 3/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.1632
- acc: 0.9517 - val_loss: 0.1437 - val_acc: 0.9574
Epoch 4/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.1266
- acc: 0.9633 - val_loss: 0.1219 - val_acc: 0.9623
Epoch 5/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.1001
- acc: 0.9706 - val_loss: 0.0997 - val_acc: 0.9697
Epoch 6/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0799
- acc: 0.9767 - val_loss: 0.0938 - val_acc: 0.9708
Epoch 7/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0651
- acc: 0.9799 - val_loss: 0.0816 - val_acc: 0.9747
Epoch 8/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0529
- acc: 0.9845 - val_loss: 0.0772 - val_acc: 0.9770
Epoch 9/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0422
- acc: 0.9881 - val_loss: 0.0754 - val_acc: 0.9764
Epoch 10/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0344
- acc: 0.9900 - val_loss: 0.0663 - val_acc: 0.9801
Epoch 11/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0278
- acc: 0.9924 - val_loss: 0.0631 - val_acc: 0.9794
Epoch 12/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0225
- acc: 0.9940 - val_loss: 0.0624 - val_acc: 0.9816
Epoch 13/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0180
- acc: 0.9955 - val_loss: 0.0632 - val_acc: 0.9802
Epoch 14/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0148
- acc: 0.9963 - val_loss: 0.0646 - val_acc: 0.9811
Epoch 15/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0115
- acc: 0.9973 - val_loss: 0.0606 - val_acc: 0.9824
Epoch 16/20
```

```
60000/60000 [==============================] - 2s 31us/step - loss: 0.0102
- acc: 0.9977 - val_loss: 0.0640 - val_acc: 0.9818
Epoch 17/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0075
- acc: 0.9985 - val_loss: 0.0674 - val_acc: 0.9808
Epoch 18/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0062
- acc: 0.9987 - val_loss: 0.0631 - val_acc: 0.9828
Epoch 19/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0042
- acc: 0.9993 - val_loss: 0.0706 - val_acc: 0.9806
Epoch 20/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0040
- acc: 0.9990 - val_loss: 0.0753 - val_acc: 0.9806
```

In [21]:

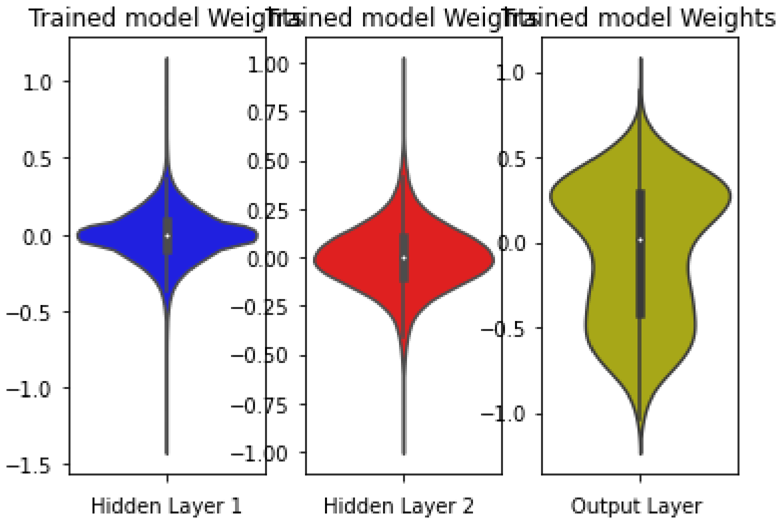
```
score = model_sigmoid.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.07527304192187294
Test accuracy: 0.9806
```



In [22]:

```
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 [23]:

```
# 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=RandomNormal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))

model_relu.summary()
```

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

Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_9 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_10 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

In [24]:

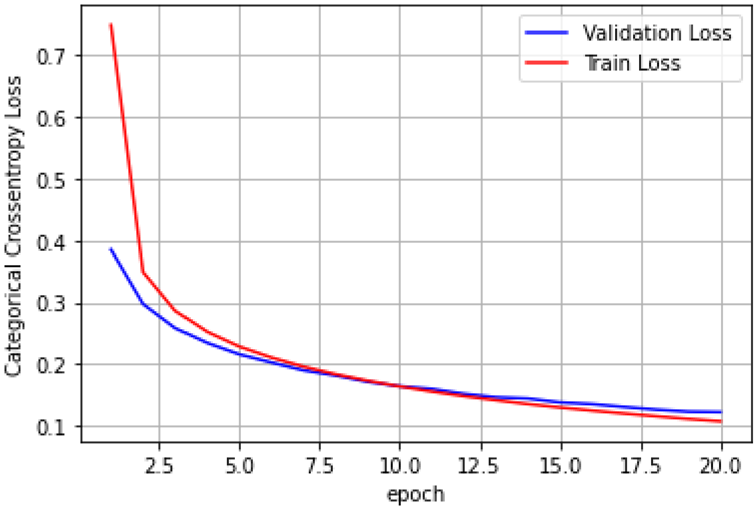
```
model_relu.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accurac
y'])

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.7488
- acc: 0.7907 - val_loss: 0.3851 - val_acc: 0.8964
Epoch 2/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3489
- acc: 0.9021 - val_loss: 0.2973 - val_acc: 0.9160
Epoch 3/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.2862
- acc: 0.9180 - val_loss: 0.2582 - val_acc: 0.9274
Epoch 4/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.2520
- acc: 0.9283 - val_loss: 0.2341 - val_acc: 0.9334
Epoch 5/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.2282
- acc: 0.9350 - val_loss: 0.2156 - val_acc: 0.9371
Epoch 6/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.2105
- acc: 0.9404 - val_loss: 0.2025 - val_acc: 0.9417
Epoch 7/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.1958
- acc: 0.9440 - val_loss: 0.1898 - val_acc: 0.9457
Epoch 8/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.1835
- acc: 0.9479 - val_loss: 0.1814 - val_acc: 0.9460
Epoch 9/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.1730
- acc: 0.9511 - val_loss: 0.1713 - val_acc: 0.9492
Epoch 10/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.1638
- acc: 0.9532 - val_loss: 0.1640 - val_acc: 0.9512
Epoch 11/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.1555
- acc: 0.9556 - val_loss: 0.1598 - val_acc: 0.9530
Epoch 12/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.1480
- acc: 0.9580 - val_loss: 0.1518 - val_acc: 0.9552
Epoch 13/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.1414
- acc: 0.9597 - val_loss: 0.1462 - val_acc: 0.9568
Epoch 14/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.1352
- acc: 0.9618 - val_loss: 0.1442 - val_acc: 0.9580
Epoch 15/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.1296
- acc: 0.9634 - val_loss: 0.1379 - val_acc: 0.9592
Epoch 16/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.1245
- acc: 0.9651 - val_loss: 0.1351 - val_acc: 0.9604
Epoch 17/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.1198
- acc: 0.9665 - val_loss: 0.1304 - val_acc: 0.9604
Epoch 18/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.1154
- acc: 0.9679 - val_loss: 0.1261 - val_acc: 0.9617
Epoch 19/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.1111
- acc: 0.9689 - val_loss: 0.1229 - val_acc: 0.9624
Epoch 20/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.1073
- acc: 0.9702 - val_loss: 0.1221 - val_acc: 0.9627
```

In [25]:

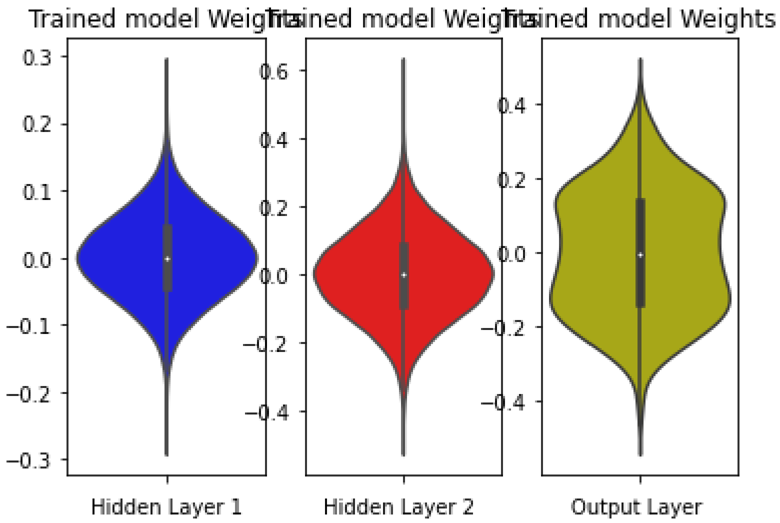
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.12213720679543913
Test accuracy: 0.9627
```



In [26]:

```
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 [27]:

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

print(model_relu.summary())

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

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

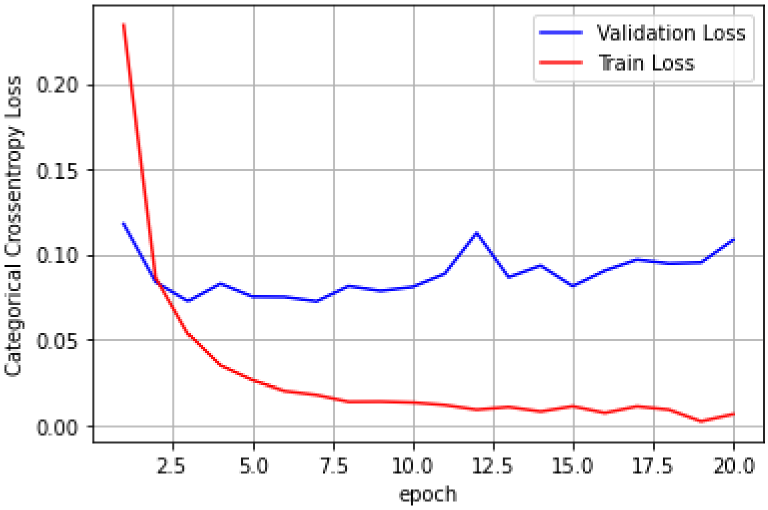
```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_11 (Dense)             (None, 512)               401920    
_________________________________________________________________
dense_12 (Dense)             (None, 128)               65664     
_________________________________________________________________
dense_13 (Dense)             (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
None
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.2344
- acc: 0.9307 - val_loss: 0.1179 - val_acc: 0.9626
Epoch 2/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0863
- acc: 0.9736 - val_loss: 0.0837 - val_acc: 0.9713
Epoch 3/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0533
- acc: 0.9838 - val_loss: 0.0725 - val_acc: 0.9762
Epoch 4/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0347
- acc: 0.9897 - val_loss: 0.0827 - val_acc: 0.9739
Epoch 5/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0261
- acc: 0.9917 - val_loss: 0.0751 - val_acc: 0.9776
Epoch 6/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.0198
- acc: 0.9937 - val_loss: 0.0749 - val_acc: 0.9785
Epoch 7/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0176
- acc: 0.9943 - val_loss: 0.0723 - val_acc: 0.9801
Epoch 8/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0136
- acc: 0.9957 - val_loss: 0.0813 - val_acc: 0.9787
Epoch 9/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0137
- acc: 0.9951 - val_loss: 0.0784 - val_acc: 0.9778
Epoch 10/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0132
- acc: 0.9956 - val_loss: 0.0808 - val_acc: 0.9801
Epoch 11/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0117
- acc: 0.9960 - val_loss: 0.0885 - val_acc: 0.9782
Epoch 12/20
60000/60000 [==============================] - 2s 29us/step - loss: 0.0089
- acc: 0.9970 - val_loss: 0.1125 - val_acc: 0.9748
Epoch 13/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0105
- acc: 0.9964 - val_loss: 0.0865 - val_acc: 0.9805
Epoch 14/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0079
- acc: 0.9972 - val_loss: 0.0933 - val_acc: 0.9800
Epoch 15/20
60000/60000 [==============================] - 2s 30us/step - loss: 0.0109
- acc: 0.9965 - val_loss: 0.0812 - val_acc: 0.9833
```

```
Epoch 16/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0070
- acc: 0.9976 - val_loss: 0.0904 - val_acc: 0.9812
Epoch 17/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0109
- acc: 0.9966 - val_loss: 0.0968 - val_acc: 0.9799
Epoch 18/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0089
- acc: 0.9970 - val_loss: 0.0945 - val_acc: 0.9803
Epoch 19/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0021
- acc: 0.9994 - val_loss: 0.0951 - val_acc: 0.9825
Epoch 20/20
60000/60000 [==============================] - 2s 31us/step - loss: 0.0063
- acc: 0.9981 - val_loss: 0.1084 - val_acc: 0.9785
```

In [28]:

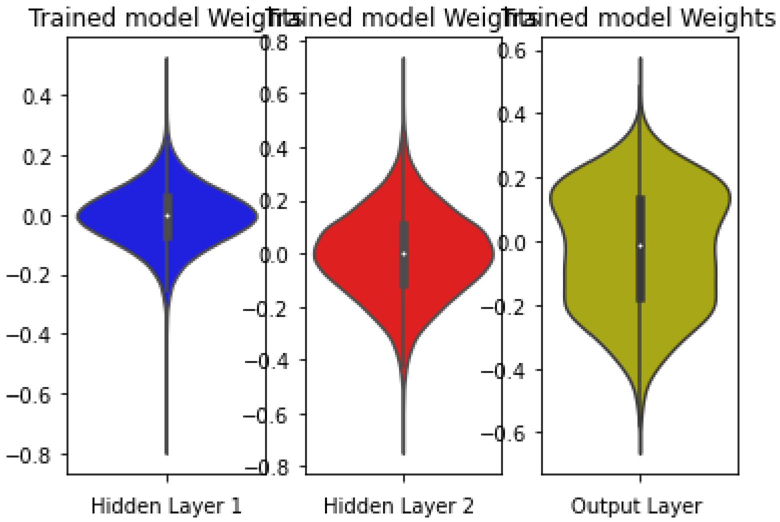
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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


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

```
Test score: 0.10837043899441291
Test accuracy: 0.9785
```



In [29]:

```
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 [30]:

```
# 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_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

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

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


model_batch.summary()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backe
nd/tensorflow_backend.py:148: The name tf.placeholder_with_default is depr
ecated. Please use tf.compat.v1.placeholder_with_default instead.

Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_14 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048      
_________________________________________________________________
dense_15 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512       
_________________________________________________________________
dense_16 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [31]:

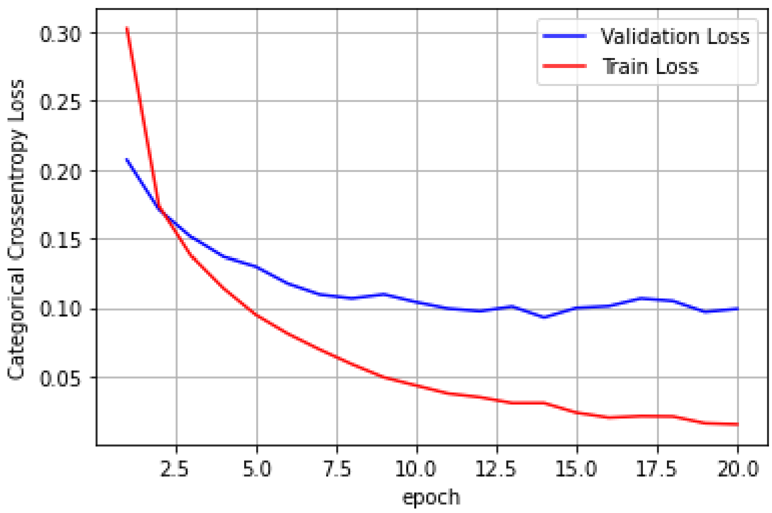
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura
cy'])

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.3026
- acc: 0.9102 - val_loss: 0.2071 - val_acc: 0.9416
Epoch 2/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.1736
- acc: 0.9497 - val_loss: 0.1710 - val_acc: 0.9504
Epoch 3/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.1374
- acc: 0.9598 - val_loss: 0.1511 - val_acc: 0.9542
Epoch 4/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.1140
- acc: 0.9656 - val_loss: 0.1368 - val_acc: 0.9589
Epoch 5/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.0949
- acc: 0.9712 - val_loss: 0.1299 - val_acc: 0.9603
Epoch 6/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0811
- acc: 0.9755 - val_loss: 0.1175 - val_acc: 0.9647
Epoch 7/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0696
- acc: 0.9780 - val_loss: 0.1096 - val_acc: 0.9661
Epoch 8/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0589
- acc: 0.9820 - val_loss: 0.1067 - val_acc: 0.9688
Epoch 9/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0493
- acc: 0.9847 - val_loss: 0.1097 - val_acc: 0.9682
Epoch 10/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0434
- acc: 0.9859 - val_loss: 0.1041 - val_acc: 0.9698
Epoch 11/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0376
- acc: 0.9879 - val_loss: 0.0994 - val_acc: 0.9697
Epoch 12/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0349
- acc: 0.9887 - val_loss: 0.0974 - val_acc: 0.9722
Epoch 13/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0307
- acc: 0.9903 - val_loss: 0.1008 - val_acc: 0.9723
Epoch 14/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0306
- acc: 0.9900 - val_loss: 0.0929 - val_acc: 0.9745
Epoch 15/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0237
- acc: 0.9924 - val_loss: 0.0998 - val_acc: 0.9732
Epoch 16/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0201
- acc: 0.9938 - val_loss: 0.1011 - val_acc: 0.9731
Epoch 17/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0211
- acc: 0.9930 - val_loss: 0.1067 - val_acc: 0.9707
Epoch 18/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0209
- acc: 0.9931 - val_loss: 0.1049 - val_acc: 0.9719
Epoch 19/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0160
- acc: 0.9951 - val_loss: 0.0969 - val_acc: 0.9737
Epoch 20/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0152
- acc: 0.9949 - val_loss: 0.0991 - val_acc: 0.9741
```

In [32]:

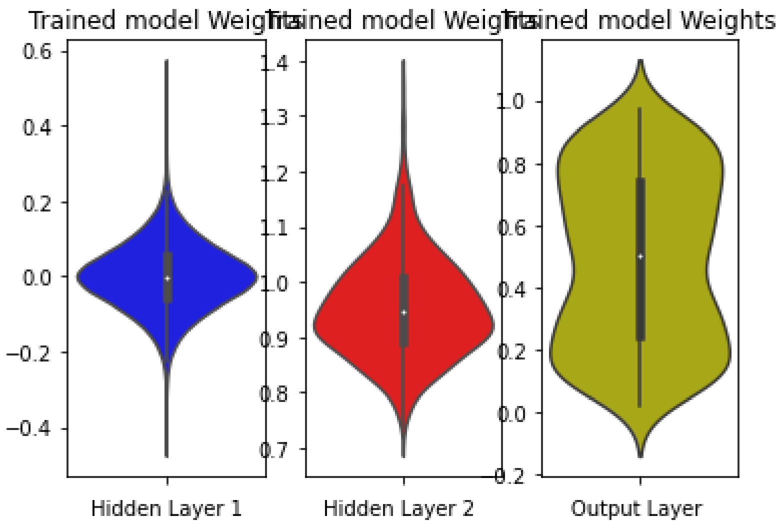
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.09913447910312971
Test accuracy: 0.9741
```



In [33]:

```
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()
```



MLP + Batch-Norm using 3 hidden Layers + AdamOptimizer

In [34]:

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

model_batch.add(Dense(200, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(170, activation='sigmoid', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_17 (Dense)             (None, 250)               196250    
_________________________________________________________________
batch_normalization_3 (Batch (None, 250)               1000      
_________________________________________________________________
dense_18 (Dense)             (None, 200)               50200     
_________________________________________________________________
batch_normalization_4 (Batch (None, 200)               800       
_________________________________________________________________
dense_19 (Dense)             (None, 170)               34170     
_________________________________________________________________
batch_normalization_5 (Batch (None, 170)               680       
_________________________________________________________________
dense_20 (Dense)             (None, 10)                1710      
=================================================================
Total params: 284,810
Trainable params: 283,570
Non-trainable params: 1,240
_________________________________________________________________
```

In [35]:

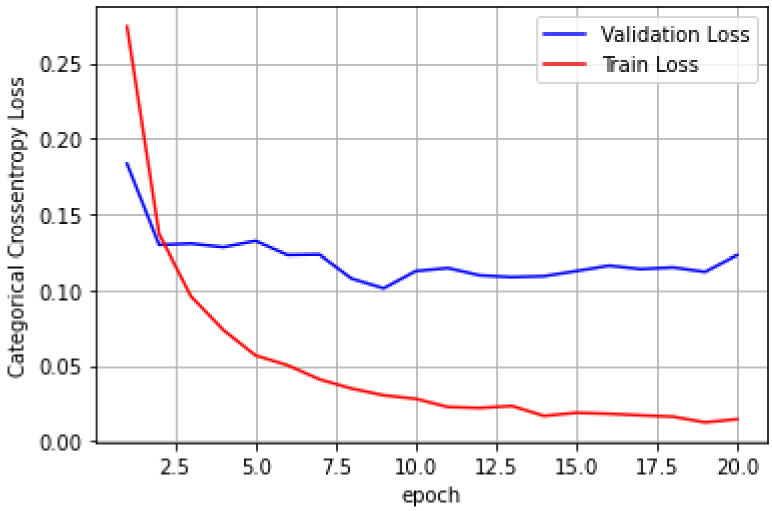
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura
cy'])

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 76us/step - loss: 0.2743
- acc: 0.9182 - val_loss: 0.1835 - val_acc: 0.9450
Epoch 2/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.1370
- acc: 0.9581 - val_loss: 0.1298 - val_acc: 0.9609
Epoch 3/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.0957
- acc: 0.9707 - val_loss: 0.1306 - val_acc: 0.9613
Epoch 4/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0737
- acc: 0.9761 - val_loss: 0.1282 - val_acc: 0.9619
Epoch 5/20
60000/60000 [==============================] - 4s 60us/step - loss: 0.0569
- acc: 0.9813 - val_loss: 0.1324 - val_acc: 0.9610
Epoch 6/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0504
- acc: 0.9826 - val_loss: 0.1231 - val_acc: 0.9630
Epoch 7/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0410
- acc: 0.9864 - val_loss: 0.1235 - val_acc: 0.9670
Epoch 8/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0350
- acc: 0.9883 - val_loss: 0.1077 - val_acc: 0.9689
Epoch 9/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0305
- acc: 0.9900 - val_loss: 0.1013 - val_acc: 0.9726
Epoch 10/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0282
- acc: 0.9907 - val_loss: 0.1127 - val_acc: 0.9695
Epoch 11/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0228
- acc: 0.9924 - val_loss: 0.1147 - val_acc: 0.9703
Epoch 12/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0221
- acc: 0.9926 - val_loss: 0.1099 - val_acc: 0.9719
Epoch 13/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0234
- acc: 0.9920 - val_loss: 0.1087 - val_acc: 0.9704
Epoch 14/20
60000/60000 [==============================] - 4s 65us/step - loss: 0.0168
- acc: 0.9942 - val_loss: 0.1093 - val_acc: 0.9729
Epoch 15/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0189
- acc: 0.9936 - val_loss: 0.1127 - val_acc: 0.9726
Epoch 16/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0183
- acc: 0.9938 - val_loss: 0.1163 - val_acc: 0.9706
Epoch 17/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.0172
- acc: 0.9941 - val_loss: 0.1140 - val_acc: 0.9736
Epoch 18/20
60000/60000 [==============================] - 4s 62us/step - loss: 0.0163
- acc: 0.9942 - val_loss: 0.1151 - val_acc: 0.9703
Epoch 19/20
60000/60000 [==============================] - 4s 64us/step - loss: 0.0126
- acc: 0.9958 - val_loss: 0.1122 - val_acc: 0.9757
Epoch 20/20
60000/60000 [==============================] - 4s 61us/step - loss: 0.0146
- acc: 0.9948 - val_loss: 0.1231 - val_acc: 0.9736
```

In [36]:

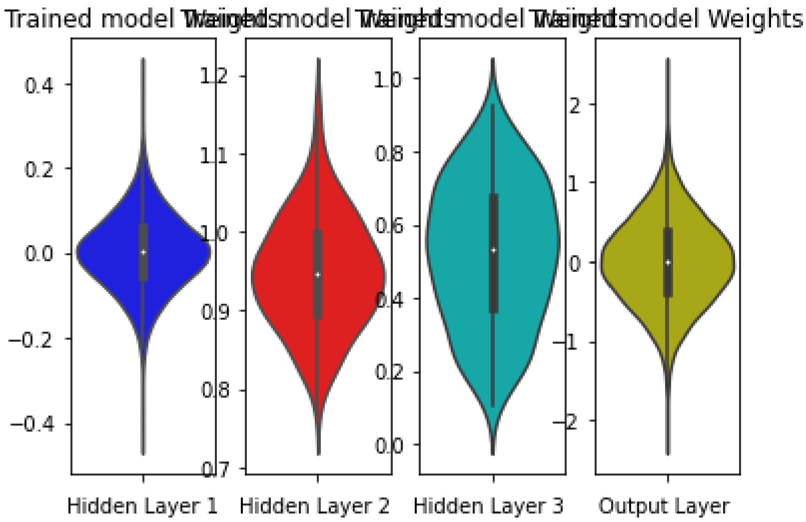
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.12309521172174281
Test accuracy: 0.9736
```



In [37]:

```
w_after = model_batch.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 4, 1)
plt.title("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='c')
plt.xlabel('Hidden Layer 3 ')

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



MLP + Batch-Norm using 5 hidden Layers + AdamOptimizer

In [38]:

```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy this 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(600, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(500, activation='sigmoid', kernel_initializer=RandomNormal(mean=
0.0, stddev=0.55, seed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(400, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(300, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(200, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_21 (Dense)             (None, 600)               471000    
_________________________________________________________________
batch_normalization_6 (Batch (None, 600)               2400      
_________________________________________________________________
dense_22 (Dense)             (None, 500)               300500    
_________________________________________________________________
batch_normalization_7 (Batch (None, 500)               2000      
_________________________________________________________________
dense_23 (Dense)             (None, 400)               200400    
_________________________________________________________________
batch_normalization_8 (Batch (None, 400)               1600      
_________________________________________________________________
dense_24 (Dense)             (None, 300)               120300    
_________________________________________________________________
batch_normalization_9 (Batch (None, 300)               1200      
_________________________________________________________________
dense_25 (Dense)             (None, 200)               60200     
_________________________________________________________________
batch_normalization_10 (Batc (None, 200)               800       
_________________________________________________________________
dense_26 (Dense)             (None, 10)                2010      
=================================================================
Total params: 1,162,410
Trainable params: 1,158,410
Non-trainable params: 4,000
_________________________________________________________________
```

In [39]:

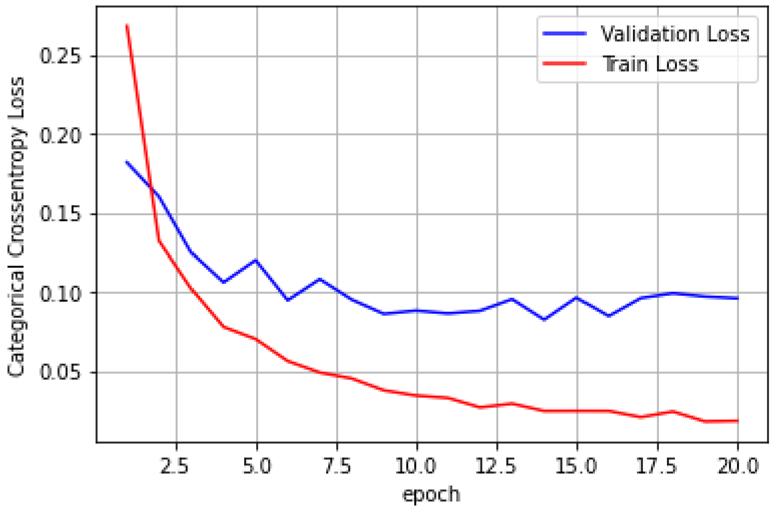
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura
cy'])

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.268
2 - acc: 0.9204 - val_loss: 0.1818 - val_acc: 0.9423
Epoch 2/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.1321
- acc: 0.9597 - val_loss: 0.1601 - val_acc: 0.9535
Epoch 3/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.1019
- acc: 0.9677 - val_loss: 0.1247 - val_acc: 0.9618
Epoch 4/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0778
- acc: 0.9753 - val_loss: 0.1060 - val_acc: 0.9686
Epoch 5/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0701
- acc: 0.9768 - val_loss: 0.1200 - val_acc: 0.9639
Epoch 6/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0562
- acc: 0.9818 - val_loss: 0.0947 - val_acc: 0.9727
Epoch 7/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0489
- acc: 0.9834 - val_loss: 0.1080 - val_acc: 0.9695
Epoch 8/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0451
- acc: 0.9853 - val_loss: 0.0951 - val_acc: 0.9730
Epoch 9/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0376
- acc: 0.9878 - val_loss: 0.0860 - val_acc: 0.9753
Epoch 10/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0343
- acc: 0.9884 - val_loss: 0.0881 - val_acc: 0.9747
Epoch 11/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0328
- acc: 0.9888 - val_loss: 0.0863 - val_acc: 0.9752
Epoch 12/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0268
- acc: 0.9911 - val_loss: 0.0880 - val_acc: 0.9760
Epoch 13/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0292
- acc: 0.9905 - val_loss: 0.0954 - val_acc: 0.9734
Epoch 14/20
60000/60000 [==============================] - 5s 86us/step - loss: 0.0245
- acc: 0.9920 - val_loss: 0.0824 - val_acc: 0.9768
Epoch 15/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0246
- acc: 0.9920 - val_loss: 0.0963 - val_acc: 0.9762
Epoch 16/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0245
- acc: 0.9917 - val_loss: 0.0847 - val_acc: 0.9776
Epoch 17/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0206
- acc: 0.9927 - val_loss: 0.0960 - val_acc: 0.9770
Epoch 18/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.0242
- acc: 0.9920 - val_loss: 0.0991 - val_acc: 0.9758
Epoch 19/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0179
- acc: 0.9937 - val_loss: 0.0971 - val_acc: 0.9759
Epoch 20/20
60000/60000 [==============================] - 5s 85us/step - loss: 0.0183
- acc: 0.9935 - val_loss: 0.0959 - val_acc: 0.9766
```

In [40]:

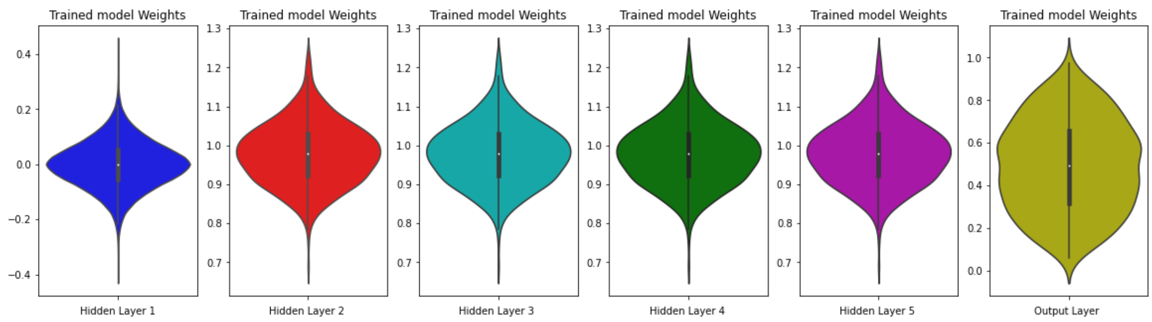
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.09593826632477576
Test accuracy: 0.9766
```



In [41]:

```
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[2].flatten().reshape(-1,1)
h4_w = w_after[2].flatten().reshape(-1,1)
h5_w = w_after[2].flatten().reshape(-1,1)
out_w = w_after[4].flatten().reshape(-1,1)


fig = plt.figure(figsize=(20,5))
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='c')
plt.xlabel('Hidden Layer 3 ')

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

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h5_w, color='m')
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()
```



MLP + Batch-Norm using 7 hidden Layers + AdamOptimizer

In [42]:

```
# 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_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(720, activation='sigmoid', kernel_initializer=RandomNormal(mean=
0.0, stddev=0.55, seed=None)) )
model_batch.add(BatchNormalization())

model_batch.add(Dense(650, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(580, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(510, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(450, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

model_batch.add(Dense(390, activation='sigmoid', input_shape=(input_dim,), kernel_initi
alizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_batch.add(BatchNormalization())

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


model_batch.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_27 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_11 (Batc (None, 512)               2048      
_________________________________________________________________
dense_28 (Dense)             (None, 720)               369360    
_________________________________________________________________
batch_normalization_12 (Batc (None, 720)               2880      
_________________________________________________________________
dense_29 (Dense)             (None, 650)               468650    
_________________________________________________________________
batch_normalization_13 (Batc (None, 650)               2600      
_________________________________________________________________
dense_30 (Dense)             (None, 580)               377580    
_________________________________________________________________
batch_normalization_14 (Batc (None, 580)               2320      
_________________________________________________________________
dense_31 (Dense)             (None, 510)               296310    
_________________________________________________________________
batch_normalization_15 (Batc (None, 510)               2040      
_________________________________________________________________
dense_32 (Dense)             (None, 450)               229950    
_________________________________________________________________
batch_normalization_16 (Batc (None, 450)               1800      
_________________________________________________________________
dense_33 (Dense)             (None, 390)               175890    
_________________________________________________________________
batch_normalization_17 (Batc (None, 390)               1560      
_________________________________________________________________
dense_34 (Dense)             (None, 10)                3910      
=================================================================
Total params: 2,338,818
Trainable params: 2,331,194
Non-trainable params: 7,624
_________________________________________________________________
```

In [43]:

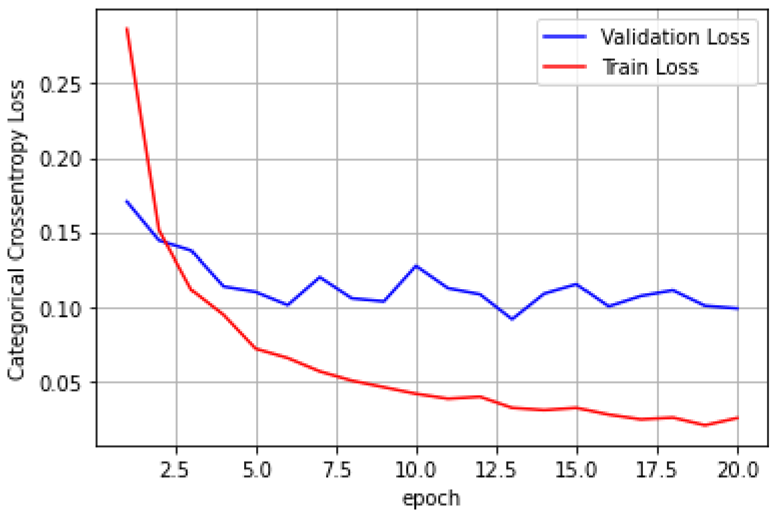
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura
cy'])

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.286
5 - acc: 0.9165 - val_loss: 0.1705 - val_acc: 0.9518
Epoch 2/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.151
3 - acc: 0.9553 - val_loss: 0.1444 - val_acc: 0.9573
Epoch 3/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.111
5 - acc: 0.9661 - val_loss: 0.1376 - val_acc: 0.9631
Epoch 4/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.094
7 - acc: 0.9707 - val_loss: 0.1137 - val_acc: 0.9671
Epoch 5/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.071
9 - acc: 0.9779 - val_loss: 0.1100 - val_acc: 0.9680
Epoch 6/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.065
6 - acc: 0.9794 - val_loss: 0.1012 - val_acc: 0.9720
Epoch 7/20
60000/60000 [==============================] - 7s 117us/step - loss: 0.056
6 - acc: 0.9824 - val_loss: 0.1200 - val_acc: 0.9682
Epoch 8/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.050
4 - acc: 0.9843 - val_loss: 0.1058 - val_acc: 0.9723
Epoch 9/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.046
0 - acc: 0.9856 - val_loss: 0.1037 - val_acc: 0.9725
Epoch 10/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.041
6 - acc: 0.9868 - val_loss: 0.1276 - val_acc: 0.9667
Epoch 11/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.038
3 - acc: 0.9882 - val_loss: 0.1126 - val_acc: 0.9716
Epoch 12/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.039
6 - acc: 0.9883 - val_loss: 0.1085 - val_acc: 0.9718
Epoch 13/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.032
1 - acc: 0.9895 - val_loss: 0.0917 - val_acc: 0.9754
Epoch 14/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.030
6 - acc: 0.9905 - val_loss: 0.1089 - val_acc: 0.9722
Epoch 15/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.032
1 - acc: 0.9896 - val_loss: 0.1152 - val_acc: 0.9716
Epoch 16/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.027
6 - acc: 0.9912 - val_loss: 0.1003 - val_acc: 0.9751
Epoch 17/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.024
4 - acc: 0.9926 - val_loss: 0.1074 - val_acc: 0.9739
Epoch 18/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.025
6 - acc: 0.9915 - val_loss: 0.1112 - val_acc: 0.9739
Epoch 19/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.020
4 - acc: 0.9937 - val_loss: 0.1008 - val_acc: 0.9770
Epoch 20/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.025
3 - acc: 0.9925 - val_loss: 0.0990 - val_acc: 0.9760
```

In [44]:

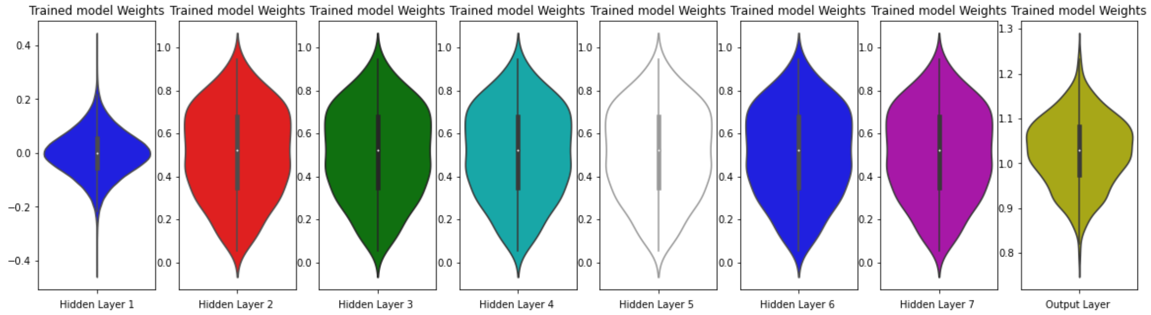
```
score = model_batch.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.09895186286373064
Test accuracy: 0.976
```



In [45]:

```
w_after = model_batch.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[4].flatten().reshape(-1,1)
h3_w = w_after[4].flatten().reshape(-1,1)
h4_w = w_after[4].flatten().reshape(-1,1)
h5_w = w_after[4].flatten().reshape(-1,1)
h6_w = w_after[4].flatten().reshape(-1,1)
h7_w = w_after[4].flatten().reshape(-1,1)
out_w = w_after[8].flatten().reshape(-1,1)


fig = plt.figure(figsize =(20,5))
plt.title("Weight matrices after model trained")
plt.subplot(1, 8, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 8, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 8, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(1, 8, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='c')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(1, 8, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h5_w, color='w')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(1, 8, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='b')
plt.xlabel('Hidden Layer 6 ')

plt.subplot(1, 8, 7)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h7_w, color='m')
plt.xlabel('Hidden Layer 7 ')

plt.subplot(1, 8, 8)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



5. MLP + 2 Dropout's + AdamOptimizer

In [46]:

```
from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='sigmoid', input_shape=(input_dim,), kernel_initia
lizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
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(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/backe
nd/tensorflow_backend.py:3733: calling dropout (from tensorflow.python.op
s.nn_ops) with keep_prob is deprecated and will be removed in a future ver
sion.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1
- keep_prob`.
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_35 (Dense)             (None, 512)               401920    
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_36 (Dense)             (None, 128)               65664     
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_37 (Dense)             (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

In [47]:

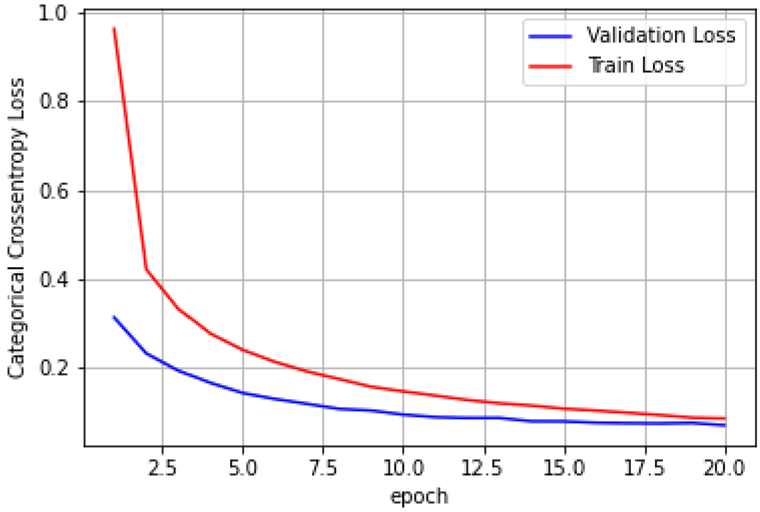
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 59us/step - loss: 0.9635
- acc: 0.6934 - val_loss: 0.3132 - val_acc: 0.9074
Epoch 2/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.4220
- acc: 0.8755 - val_loss: 0.2321 - val_acc: 0.9292
Epoch 3/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3325
- acc: 0.9022 - val_loss: 0.1930 - val_acc: 0.9415
Epoch 4/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.2764
- acc: 0.9184 - val_loss: 0.1652 - val_acc: 0.9481
Epoch 5/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.2397
- acc: 0.9291 - val_loss: 0.1425 - val_acc: 0.9555
Epoch 6/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.2129
- acc: 0.9382 - val_loss: 0.1291 - val_acc: 0.9598
Epoch 7/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.1912
- acc: 0.9437 - val_loss: 0.1180 - val_acc: 0.9628
Epoch 8/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.1740
- acc: 0.9498 - val_loss: 0.1067 - val_acc: 0.9651
Epoch 9/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.1566
- acc: 0.9541 - val_loss: 0.1030 - val_acc: 0.9678
Epoch 10/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.1461
- acc: 0.9571 - val_loss: 0.0938 - val_acc: 0.9708
Epoch 11/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1367
- acc: 0.9596 - val_loss: 0.0880 - val_acc: 0.9722
Epoch 12/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.1268
- acc: 0.9619 - val_loss: 0.0862 - val_acc: 0.9727
Epoch 13/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1193
- acc: 0.9642 - val_loss: 0.0863 - val_acc: 0.9745
Epoch 14/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1141
- acc: 0.9657 - val_loss: 0.0787 - val_acc: 0.9762
Epoch 15/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.1072
- acc: 0.9681 - val_loss: 0.0785 - val_acc: 0.9764
Epoch 16/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1029
- acc: 0.9693 - val_loss: 0.0754 - val_acc: 0.9775
Epoch 17/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0979
- acc: 0.9711 - val_loss: 0.0743 - val_acc: 0.9772
Epoch 18/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0927
- acc: 0.9720 - val_loss: 0.0739 - val_acc: 0.9773
Epoch 19/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.0868
- acc: 0.9738 - val_loss: 0.0749 - val_acc: 0.9777
Epoch 20/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.0850
- acc: 0.9741 - val_loss: 0.0698 - val_acc: 0.9789
```

In [48]:

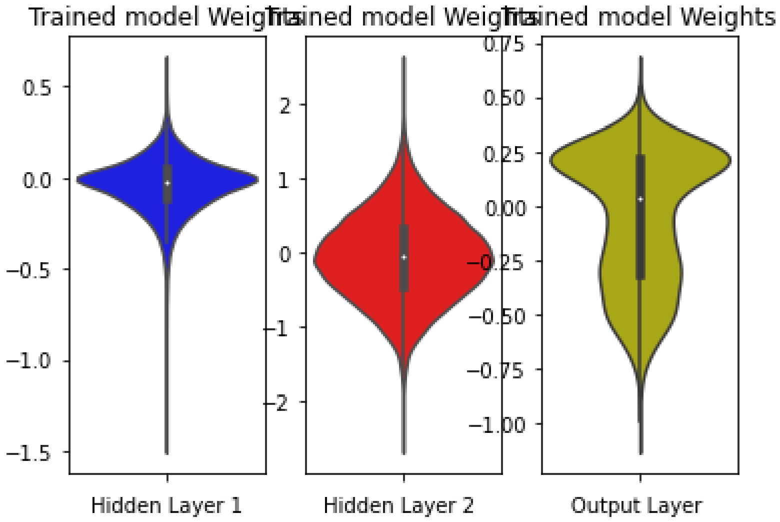
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.06975141780656995
Test accuracy: 0.9789
```



In [49]:

```
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()
```



1. MLP + 3 Dropout's + AdamOptimizer

In [55]:

```
from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(Dropout(0.5))

model_drop.add(Dense(496, activation='relu', kernel_initializer=RandomNormal(mean=0.0,
stddev=0.55, seed=None)) )
model_drop.add(Dropout(0.7))

model_drop.add(Dense(279, activation='relu', input_shape=(input_dim,), kernel_initializ
er=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(Dropout(0.5))

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

model_drop.summary()
```

```
WARNING:tensorflow:Large dropout rate: 0.7 (>0.5). In TensorFlow 2.x, drop
out() uses dropout rate instead of keep_prob. Please ensure that this is i
ntended.
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_42 (Dense)             (None, 512)               401920    
_________________________________________________________________
dropout_6 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_43 (Dense)             (None, 496)               254448    
_________________________________________________________________
dropout_7 (Dropout)          (None, 496)               0         
_________________________________________________________________
dense_44 (Dense)             (None, 279)               138663    
_________________________________________________________________
dropout_8 (Dropout)          (None, 279)               0         
_________________________________________________________________
dense_45 (Dense)             (None, 10)                2800      
=================================================================
Total params: 797,831
Trainable params: 797,831
Non-trainable params: 0
_________________________________________________________________
```

In [56]:

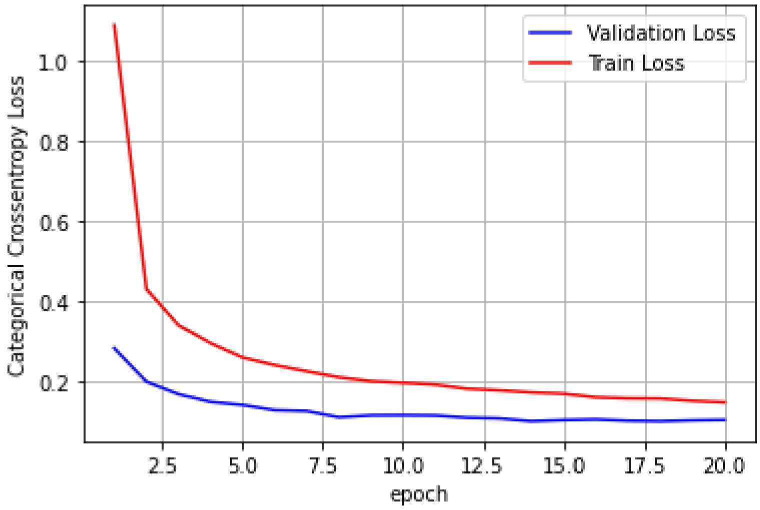
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 4s 66us/step - loss: 1.0892
- acc: 0.6844 - val_loss: 0.2835 - val_acc: 0.9236
Epoch 2/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.4311
- acc: 0.8779 - val_loss: 0.2002 - val_acc: 0.9486
Epoch 3/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.3407
- acc: 0.9052 - val_loss: 0.1691 - val_acc: 0.9541
Epoch 4/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.2960
- acc: 0.9193 - val_loss: 0.1497 - val_acc: 0.9589
Epoch 5/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.2601
- acc: 0.9295 - val_loss: 0.1421 - val_acc: 0.9587
Epoch 6/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.2414
- acc: 0.9342 - val_loss: 0.1290 - val_acc: 0.9637
Epoch 7/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.2259
- acc: 0.9382 - val_loss: 0.1269 - val_acc: 0.9641
Epoch 8/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.2111
- acc: 0.9426 - val_loss: 0.1115 - val_acc: 0.9670
Epoch 9/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.2012
- acc: 0.9452 - val_loss: 0.1162 - val_acc: 0.9647
Epoch 10/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1971
- acc: 0.9460 - val_loss: 0.1165 - val_acc: 0.9668
Epoch 11/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.1928
- acc: 0.9484 - val_loss: 0.1159 - val_acc: 0.9673
Epoch 12/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.1825
- acc: 0.9514 - val_loss: 0.1104 - val_acc: 0.9709
Epoch 13/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1785
- acc: 0.9520 - val_loss: 0.1085 - val_acc: 0.9699
Epoch 14/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.1734
- acc: 0.9539 - val_loss: 0.1016 - val_acc: 0.9726
Epoch 15/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.1701
- acc: 0.9539 - val_loss: 0.1045 - val_acc: 0.9715
Epoch 16/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1611
- acc: 0.9581 - val_loss: 0.1061 - val_acc: 0.9714
Epoch 17/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1587
- acc: 0.9571 - val_loss: 0.1027 - val_acc: 0.9712
Epoch 18/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.1581
- acc: 0.9574 - val_loss: 0.1015 - val_acc: 0.9741
Epoch 19/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.1522
- acc: 0.9591 - val_loss: 0.1037 - val_acc: 0.9738
Epoch 20/20
60000/60000 [==============================] - 2s 38us/step - loss: 0.1488
- acc: 0.9605 - val_loss: 0.1049 - val_acc: 0.9747
```

In [57]:

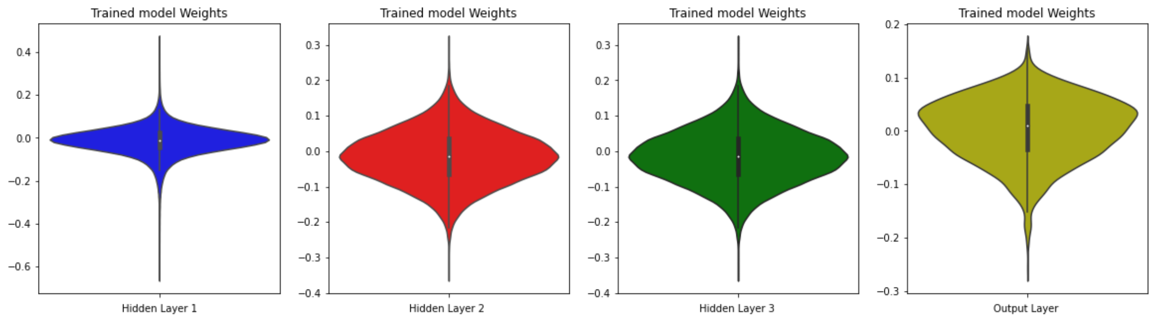
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.10485141305003781
Test accuracy: 0.9747
```



In [58]:

```
w_after = model_drop.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[4].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(figsize=(20,5))
plt.title("Weight matrices after model trained")
plt.subplot(1, 4, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

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

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



1. MLP + Batch-norm + 5 Dropout's + AdamOptimizer

In [60]:

```
from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(420, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(Dropout(0.4))

model_drop.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(Dropout(0.8))

model_drop.add(Dense(638, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(Dropout(0.25))

model_drop.add(Dense(381, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(Dropout(0.17))

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

model_drop.summary()
```

```
WARNING:tensorflow:Large dropout rate: 0.8 (>0.5). In TensorFlow 2.x, drop
out() uses dropout rate instead of keep_prob. Please ensure that this is i
ntended.
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_52 (Dense)             (None, 420)               329700    
_________________________________________________________________
dropout_14 (Dropout)         (None, 420)               0         
_________________________________________________________________
dense_53 (Dense)             (None, 128)               53888     
_________________________________________________________________
dropout_15 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_54 (Dense)             (None, 512)               66048     
_________________________________________________________________
dropout_16 (Dropout)         (None, 512)               0         
_________________________________________________________________
dense_55 (Dense)             (None, 638)               327294    
_________________________________________________________________
dropout_17 (Dropout)         (None, 638)               0         
_________________________________________________________________
dense_56 (Dense)             (None, 381)               243459    
_________________________________________________________________
dropout_18 (Dropout)         (None, 381)               0         
_________________________________________________________________
dense_57 (Dense)             (None, 10)                3820      
=================================================================
Total params: 1,024,209
Trainable params: 1,024,209
Non-trainable params: 0
_________________________________________________________________
```

In [61]:

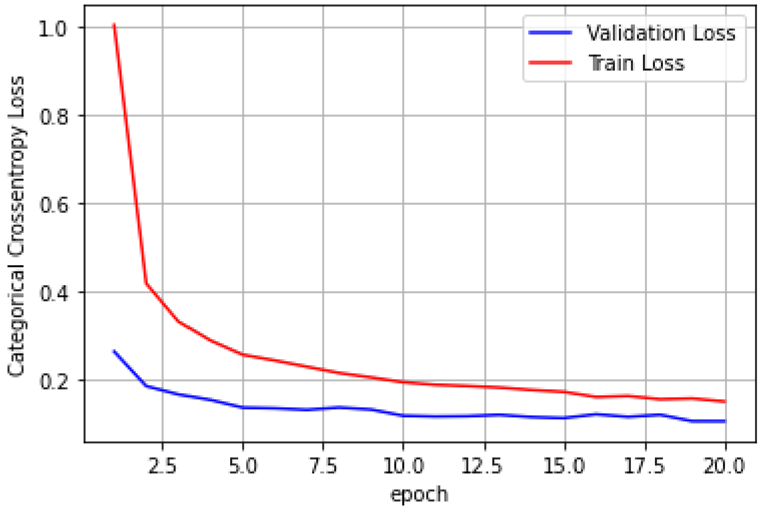
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 79us/step - loss: 1.0027
- acc: 0.6563 - val_loss: 0.2651 - val_acc: 0.9257
Epoch 2/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.4191
- acc: 0.8818 - val_loss: 0.1868 - val_acc: 0.9475
Epoch 3/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.3330
- acc: 0.9082 - val_loss: 0.1680 - val_acc: 0.9538
Epoch 4/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.2898
- acc: 0.9219 - val_loss: 0.1554 - val_acc: 0.9593
Epoch 5/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.2577
- acc: 0.9315 - val_loss: 0.1381 - val_acc: 0.9618
Epoch 6/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.2448
- acc: 0.9360 - val_loss: 0.1366 - val_acc: 0.9634
Epoch 7/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.2306
- acc: 0.9408 - val_loss: 0.1331 - val_acc: 0.9651
Epoch 8/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.2165
- acc: 0.9441 - val_loss: 0.1384 - val_acc: 0.9652
Epoch 9/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.2065
- acc: 0.9477 - val_loss: 0.1337 - val_acc: 0.9688
Epoch 10/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.1955
- acc: 0.9492 - val_loss: 0.1197 - val_acc: 0.9688
Epoch 11/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.1898
- acc: 0.9532 - val_loss: 0.1180 - val_acc: 0.9704
Epoch 12/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.1870
- acc: 0.9529 - val_loss: 0.1187 - val_acc: 0.9703
Epoch 13/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.1836
- acc: 0.9540 - val_loss: 0.1218 - val_acc: 0.9697
Epoch 14/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.1781
- acc: 0.9559 - val_loss: 0.1167 - val_acc: 0.9719
Epoch 15/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.1739
- acc: 0.9565 - val_loss: 0.1144 - val_acc: 0.9716
Epoch 16/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.1622
- acc: 0.9591 - val_loss: 0.1230 - val_acc: 0.9737
Epoch 17/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.1644
- acc: 0.9587 - val_loss: 0.1172 - val_acc: 0.9719
Epoch 18/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1570
- acc: 0.9606 - val_loss: 0.1219 - val_acc: 0.9700
Epoch 19/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.1588
- acc: 0.9593 - val_loss: 0.1074 - val_acc: 0.9739
Epoch 20/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.1519
- acc: 0.9630 - val_loss: 0.1073 - val_acc: 0.9740
```

In [62]:

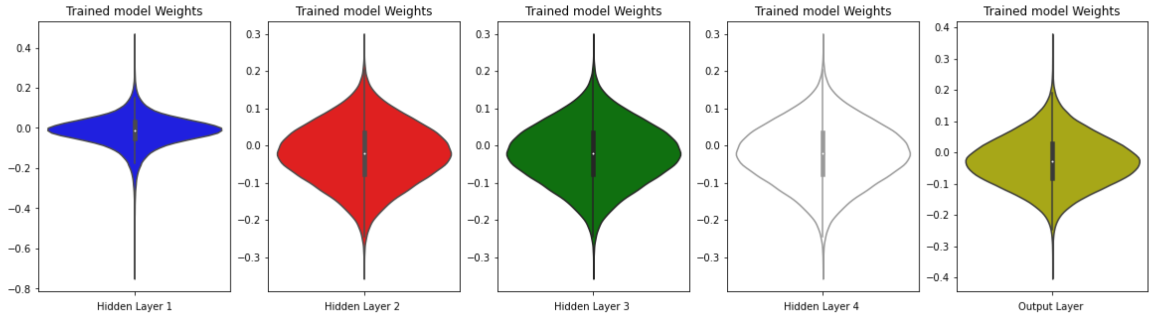
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.10725452537201345
Test accuracy: 0.974
```



In [63]:

```
w_after = model_drop.get_weights()

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


fig = plt.figure(figsize=(20,5))
plt.title("Weight matrices after model trained")
plt.subplot(1, 5, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 5, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 5, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='g')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(1,5, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='w')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(1, 5, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



6. MLP + Batch-norm + Dropout + AdamOptimizer

In [64]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='sigmoid', input_shape=(input_dim,), kernel_initia
lizer=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()
```

```
Model: "sequential_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_58 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_18 (Batc (None, 512)               2048      
_________________________________________________________________
dropout_19 (Dropout)         (None, 512)               0         
_________________________________________________________________
dense_59 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_19 (Batc (None, 128)               512       
_________________________________________________________________
dropout_20 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_60 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [65]:

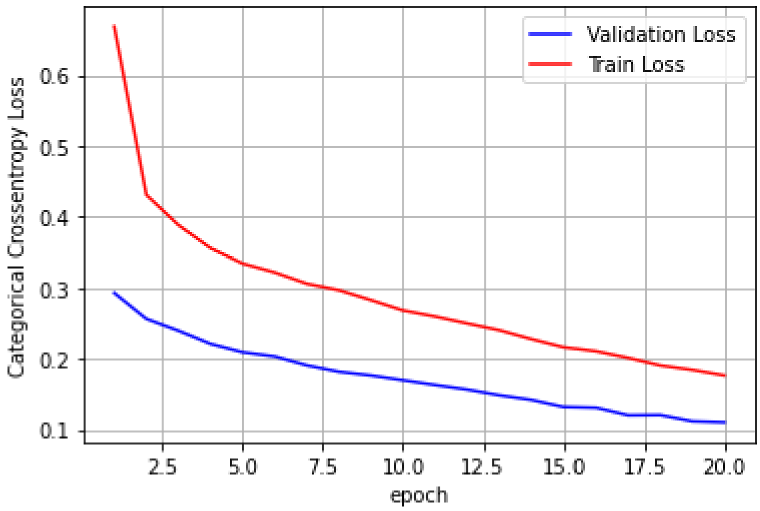
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.6686
- acc: 0.7923 - val_loss: 0.2930 - val_acc: 0.9139
Epoch 2/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.4310
- acc: 0.8690 - val_loss: 0.2566 - val_acc: 0.9244
Epoch 3/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.3884
- acc: 0.8829 - val_loss: 0.2397 - val_acc: 0.9287
Epoch 4/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.3559
- acc: 0.8929 - val_loss: 0.2211 - val_acc: 0.9332
Epoch 5/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.3343
- acc: 0.8995 - val_loss: 0.2093 - val_acc: 0.9383
Epoch 6/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.3218
- acc: 0.9037 - val_loss: 0.2034 - val_acc: 0.9403
Epoch 7/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.3060
- acc: 0.9067 - val_loss: 0.1909 - val_acc: 0.9440
Epoch 8/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.2969
- acc: 0.9093 - val_loss: 0.1818 - val_acc: 0.9460
Epoch 9/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.2830
- acc: 0.9147 - val_loss: 0.1766 - val_acc: 0.9468
Epoch 10/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.2684
- acc: 0.9189 - val_loss: 0.1699 - val_acc: 0.9491
Epoch 11/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.2599
- acc: 0.9216 - val_loss: 0.1631 - val_acc: 0.9502
Epoch 12/20
60000/60000 [==============================] - 3s 57us/step - loss: 0.2501
- acc: 0.9237 - val_loss: 0.1567 - val_acc: 0.9530
Epoch 13/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.2405
- acc: 0.9279 - val_loss: 0.1487 - val_acc: 0.9557
Epoch 14/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.2280
- acc: 0.9304 - val_loss: 0.1421 - val_acc: 0.9572
Epoch 15/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.2165
- acc: 0.9338 - val_loss: 0.1322 - val_acc: 0.9600
Epoch 16/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.2107
- acc: 0.9369 - val_loss: 0.1310 - val_acc: 0.9592
Epoch 17/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.2013
- acc: 0.9403 - val_loss: 0.1204 - val_acc: 0.9642
Epoch 18/20
60000/60000 [==============================] - 3s 55us/step - loss: 0.1907
- acc: 0.9423 - val_loss: 0.1207 - val_acc: 0.9638
Epoch 19/20
60000/60000 [==============================] - 3s 56us/step - loss: 0.1845
- acc: 0.9443 - val_loss: 0.1119 - val_acc: 0.9671
Epoch 20/20
60000/60000 [==============================] - 3s 54us/step - loss: 0.1766
- acc: 0.9464 - val_loss: 0.1105 - val_acc: 0.9670
```

In [66]:

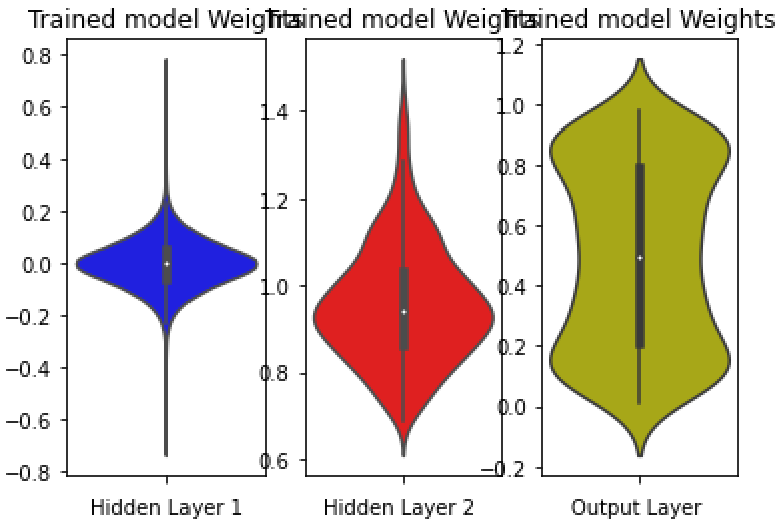
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.11045308456420898
Test accuracy: 0.967
```



In [67]:

```
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 +Bath-norm + 7 Dropout's + AdamOptimizer

In [68]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializ
er=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='relu', kernel_initializer=RandomNormal(mean=0.0,
stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.7))

model_drop.add(Dense(346, activation='relu', input_shape=(input_dim,),kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.039, seed = None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.63))

model_drop.add(Dense(496, activation='relu', input_shape=(input_dim,),kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.039, seed = None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.57))

model_drop.add(Dense(639, activation='relu', input_shape=(input_dim,),kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.039, seed = None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.19))

model_drop.add(Dense(99, 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(0.87))

model_drop.add(Dense(750, activation='relu', input_shape=(input_dim,),kernel_initialize
r=RandomNormal(mean=0.0, stddev=0.039, seed = None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.1))

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

model_drop.summary()
```

```
WARNING:tensorflow:Large dropout rate: 0.7 (>0.5). In TensorFlow 2.x, drop
out() uses dropout rate instead of keep_prob. Please ensure that this is i
ntended.
Model: "sequential_16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_61 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_20 (Batc (None, 512)               2048      
_________________________________________________________________
dropout_21 (Dropout)         (None, 512)               0         
_________________________________________________________________
dense_62 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_21 (Batc (None, 128)               512       
_________________________________________________________________
dropout_22 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_63 (Dense)             (None, 346)               44634     
_________________________________________________________________
batch_normalization_22 (Batc (None, 346)               1384      
_________________________________________________________________
dropout_23 (Dropout)         (None, 346)               0         
_________________________________________________________________
dense_64 (Dense)             (None, 496)               172112    
_________________________________________________________________
batch_normalization_23 (Batc (None, 496)               1984      
_________________________________________________________________
dropout_24 (Dropout)         (None, 496)               0         
_________________________________________________________________
dense_65 (Dense)             (None, 639)               317583    
_________________________________________________________________
batch_normalization_24 (Batc (None, 639)               2556      
_________________________________________________________________
dropout_25 (Dropout)         (None, 639)               0         
_________________________________________________________________
dense_66 (Dense)             (None, 99)                63360     
_________________________________________________________________
batch_normalization_25 (Batc (None, 99)                396       
_________________________________________________________________
dropout_26 (Dropout)         (None, 99)                0         
_________________________________________________________________
dense_67 (Dense)             (None, 750)               75000     
_________________________________________________________________
batch_normalization_26 (Batc (None, 750)               3000      
_________________________________________________________________
dropout_27 (Dropout)         (None, 750)               0         
_________________________________________________________________
dense_68 (Dense)             (None, 10)                7510      
=================================================================
Total params: 1,159,663
Trainable params: 1,153,723
Non-trainable params: 5,940
_________________________________________________________________
```

In [69]:

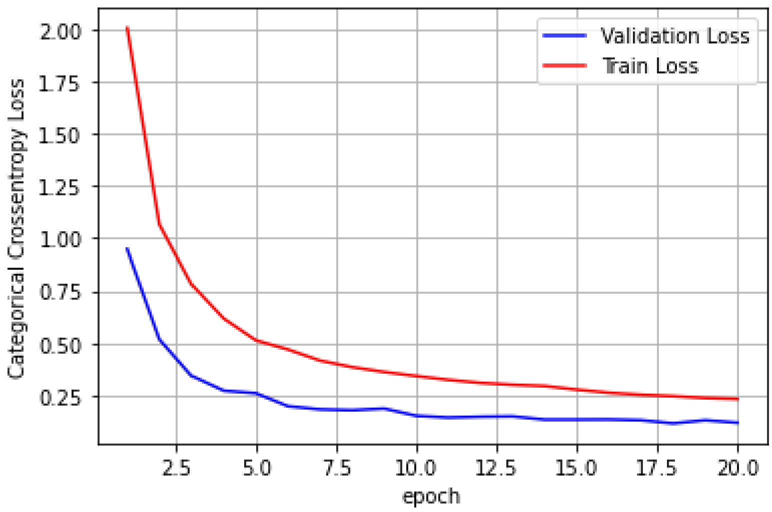
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 11s 181us/step - loss: 2.00
71 - acc: 0.2829 - val_loss: 0.9487 - val_acc: 0.6370
Epoch 2/20
60000/60000 [==============================] - 7s 123us/step - loss: 1.067
1 - acc: 0.6132 - val_loss: 0.5183 - val_acc: 0.7925
Epoch 3/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.780
5 - acc: 0.7487 - val_loss: 0.3428 - val_acc: 0.9032
Epoch 4/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.617
2 - acc: 0.8172 - val_loss: 0.2718 - val_acc: 0.9270
Epoch 5/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.513
1 - acc: 0.8576 - val_loss: 0.2597 - val_acc: 0.9339
Epoch 6/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.469
3 - acc: 0.8768 - val_loss: 0.1974 - val_acc: 0.9508
Epoch 7/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.416
0 - acc: 0.8938 - val_loss: 0.1824 - val_acc: 0.9552
Epoch 8/20
60000/60000 [==============================] - 7s 120us/step - loss: 0.384
8 - acc: 0.9035 - val_loss: 0.1783 - val_acc: 0.9594
Epoch 9/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.361
5 - acc: 0.9106 - val_loss: 0.1863 - val_acc: 0.9590
Epoch 10/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.342
1 - acc: 0.9140 - val_loss: 0.1516 - val_acc: 0.9649
Epoch 11/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.323
8 - acc: 0.9216 - val_loss: 0.1432 - val_acc: 0.9669
Epoch 12/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.309
3 - acc: 0.9256 - val_loss: 0.1470 - val_acc: 0.9661
Epoch 13/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.300
2 - acc: 0.9263 - val_loss: 0.1489 - val_acc: 0.9671
Epoch 14/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.294
3 - acc: 0.9298 - val_loss: 0.1334 - val_acc: 0.9700
Epoch 15/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.277
0 - acc: 0.9349 - val_loss: 0.1334 - val_acc: 0.9714
Epoch 16/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.261
8 - acc: 0.9384 - val_loss: 0.1340 - val_acc: 0.9713
Epoch 17/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.252
1 - acc: 0.9404 - val_loss: 0.1304 - val_acc: 0.9715
Epoch 18/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.245
6 - acc: 0.9422 - val_loss: 0.1152 - val_acc: 0.9742
Epoch 19/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.236
8 - acc: 0.9438 - val_loss: 0.1303 - val_acc: 0.9726
Epoch 20/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.232
2 - acc: 0.9459 - val_loss: 0.1185 - val_acc: 0.9742
```

In [70]:

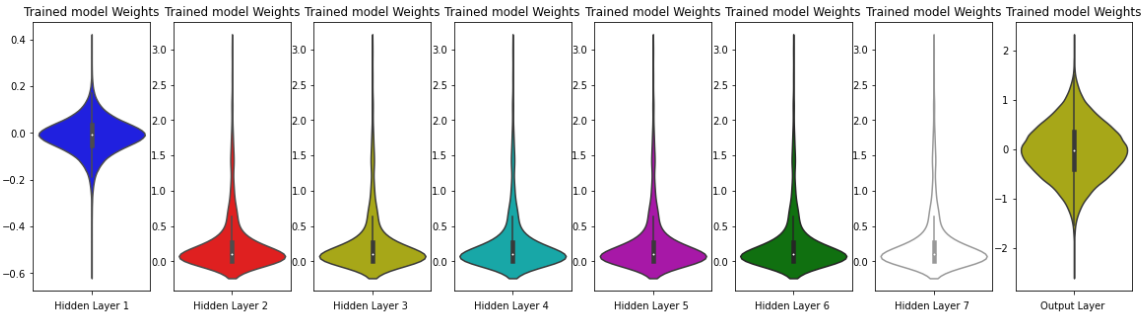
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.1184512525127735
Test accuracy: 0.9742
```



In [71]:

```
w_after = model_drop.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[4].flatten().reshape(-1,1)
h3_w = w_after[4].flatten().reshape(-1,1)
h4_w = w_after[4].flatten().reshape(-1,1)
h5_w = w_after[4].flatten().reshape(-1,1)
h6_w = w_after[4].flatten().reshape(-1,1)
h7_w = w_after[4].flatten().reshape(-1,1)
out_w = w_after[6].flatten().reshape(-1,1)


fig = plt.figure(figsize=(20,5))
plt.title("Weight matrices after model trained")
plt.subplot(1, 8, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

plt.subplot(1, 8, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 2 ')

plt.subplot(1, 8, 3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h3_w, color='y')
plt.xlabel('Hidden Layer 3 ')

plt.subplot(1, 8, 4)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h4_w, color='c')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(1, 8, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h5_w, color='m')
plt.xlabel('Hidden Layer 5 ')

plt.subplot(1, 8, 6)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h6_w, color='g')
plt.xlabel('Hidden Layer 6 ')

plt.subplot(1, 8, 7)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h7_w, color='w')
plt.xlabel('Hidden Layer 7 ')

plt.subplot(1, 8, 8)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='y')
plt.xlabel('Output Layer ')
plt.show()
```



MLP + Batch-norm+ 3 Dropout's + AdamOptimizer

In [72]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(624, activation='sigmoid', input_shape=(input_dim,), kernel_initia
lizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(370, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.
0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.4))

model_drop.add(Dense(82, activation='sigmoid', input_shape=(input_dim,),kernel_initiali
zer=RandomNormal(mean=0.0, stddev=0.039, seed = None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.3))

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

model_drop.summary()
```

```
Model: "sequential_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_69 (Dense)             (None, 624)               489840    
_________________________________________________________________
batch_normalization_27 (Batc (None, 624)               2496      
_________________________________________________________________
dropout_28 (Dropout)         (None, 624)               0         
_________________________________________________________________
dense_70 (Dense)             (None, 370)               231250    
_________________________________________________________________
batch_normalization_28 (Batc (None, 370)               1480      
_________________________________________________________________
dropout_29 (Dropout)         (None, 370)               0         
_________________________________________________________________
dense_71 (Dense)             (None, 82)                30422     
_________________________________________________________________
batch_normalization_29 (Batc (None, 82)                328       
_________________________________________________________________
dropout_30 (Dropout)         (None, 82)                0         
_________________________________________________________________
dense_72 (Dense)             (None, 10)                830       
=================================================================
Total params: 756,646
Trainable params: 754,494
Non-trainable params: 2,152
_________________________________________________________________
```

In [73]:

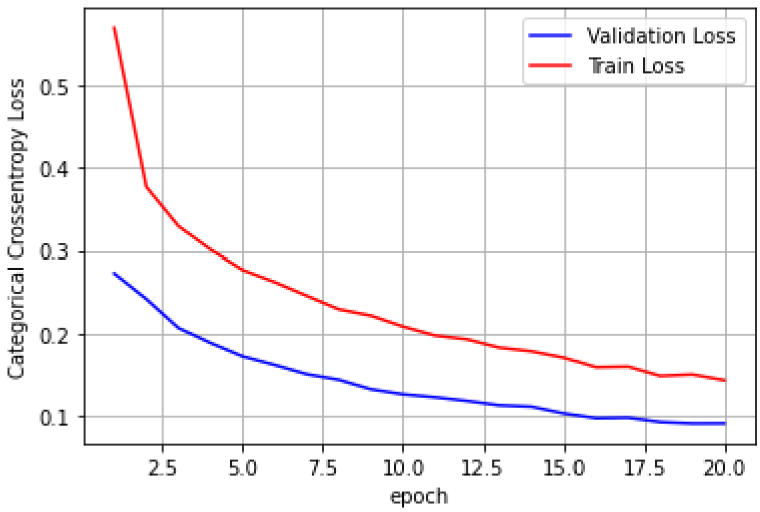
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 116us/step - loss: 0.568
7 - acc: 0.8232 - val_loss: 0.2728 - val_acc: 0.9202
Epoch 2/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.3768
- acc: 0.8873 - val_loss: 0.2421 - val_acc: 0.9277
Epoch 3/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.3294
- acc: 0.9004 - val_loss: 0.2071 - val_acc: 0.9396
Epoch 4/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.3011
- acc: 0.9099 - val_loss: 0.1889 - val_acc: 0.9442
Epoch 5/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.2769
- acc: 0.9174 - val_loss: 0.1728 - val_acc: 0.9474
Epoch 6/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.2623
- acc: 0.9220 - val_loss: 0.1624 - val_acc: 0.9520
Epoch 7/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.2459
- acc: 0.9254 - val_loss: 0.1512 - val_acc: 0.9540
Epoch 8/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.2297
- acc: 0.9312 - val_loss: 0.1445 - val_acc: 0.9558
Epoch 9/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.2220
- acc: 0.9339 - val_loss: 0.1330 - val_acc: 0.9600
Epoch 10/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.2088
- acc: 0.9373 - val_loss: 0.1269 - val_acc: 0.9622
Epoch 11/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.1980
- acc: 0.9405 - val_loss: 0.1233 - val_acc: 0.9626
Epoch 12/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1934
- acc: 0.9421 - val_loss: 0.1189 - val_acc: 0.9639
Epoch 13/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.1834
- acc: 0.9449 - val_loss: 0.1136 - val_acc: 0.9656
Epoch 14/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1790
- acc: 0.9465 - val_loss: 0.1120 - val_acc: 0.9663
Epoch 15/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.1714
- acc: 0.9490 - val_loss: 0.1038 - val_acc: 0.9682
Epoch 16/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1597
- acc: 0.9521 - val_loss: 0.0981 - val_acc: 0.9708
Epoch 17/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1604
- acc: 0.9519 - val_loss: 0.0987 - val_acc: 0.9715
Epoch 18/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1491
- acc: 0.9545 - val_loss: 0.0934 - val_acc: 0.9739
Epoch 19/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.1509
- acc: 0.9539 - val_loss: 0.0916 - val_acc: 0.9730
Epoch 20/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.1439
- acc: 0.9557 - val_loss: 0.0917 - val_acc: 0.9726
```

In [74]:

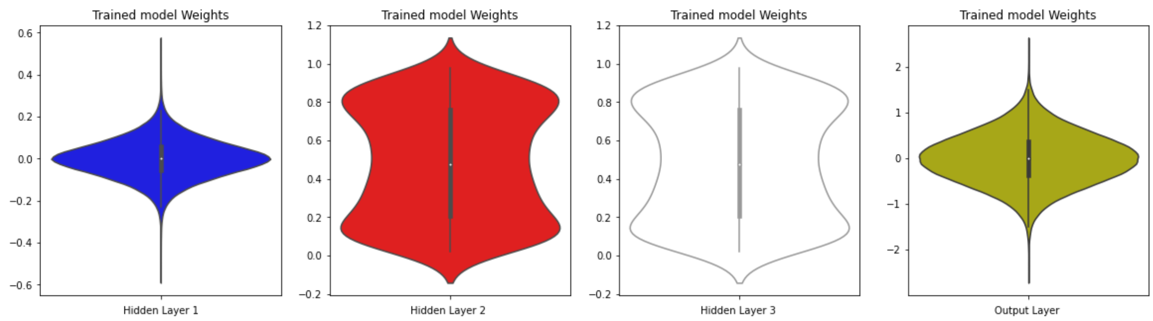
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, ve
rbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.09171288008186966
Test accuracy: 0.9726
```



In [75]:

```
w_after = model_drop.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[4].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(figsize=(20,5))
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='w')
plt.xlabel('Hidden Layer 3 ')

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



MLP +Batch-norm + 5 Dropout's + AdamOptimizer

In [76]:

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

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(710, activation='sigmoid', input_shape=(input_dim,), kernel_initia
lizer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(623, activation='sigmoid', kernel_initializer=RandomNormal(mean=0.
0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.7))

model_drop.add(Dense(548, activation='sigmoid', input_shape=(input_dim,),kernel_initial
izer=RandomNormal(mean=0.0, stddev=0.039, seed = None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.8))

model_drop.add(Dense(475, activation='sigmoid', input_shape=(input_dim,),kernel_initial
izer=RandomNormal(mean=0.0, stddev=0.039, seed = None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.2))

model_drop.add(Dense(316, activation='sigmoid', input_shape=(input_dim,),kernel_initial
izer=RandomNormal(mean=0.0, stddev=0.039, seed = None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.37))

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

model_drop.summary()
```

```
Model: "sequential_18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_73 (Dense)             (None, 710)               557350    
_________________________________________________________________
batch_normalization_30 (Batc (None, 710)               2840      
_________________________________________________________________
dropout_31 (Dropout)         (None, 710)               0         
_________________________________________________________________
dense_74 (Dense)             (None, 623)               442953    
_________________________________________________________________
batch_normalization_31 (Batc (None, 623)               2492      
_________________________________________________________________
dropout_32 (Dropout)         (None, 623)               0         
_________________________________________________________________
dense_75 (Dense)             (None, 548)               341952    
_________________________________________________________________
batch_normalization_32 (Batc (None, 548)               2192      
_________________________________________________________________
dropout_33 (Dropout)         (None, 548)               0         
_________________________________________________________________
dense_76 (Dense)             (None, 475)               260775    
_________________________________________________________________
batch_normalization_33 (Batc (None, 475)               1900      
_________________________________________________________________
dropout_34 (Dropout)         (None, 475)               0         
_________________________________________________________________
dense_77 (Dense)             (None, 316)               150416    
_________________________________________________________________
batch_normalization_34 (Batc (None, 316)               1264      
_________________________________________________________________
dropout_35 (Dropout)         (None, 316)               0         
_________________________________________________________________
dense_78 (Dense)             (None, 10)                3170      
=================================================================
Total params: 1,767,304
Trainable params: 1,761,960
Non-trainable params: 5,344
_________________________________________________________________
```

In [77]:

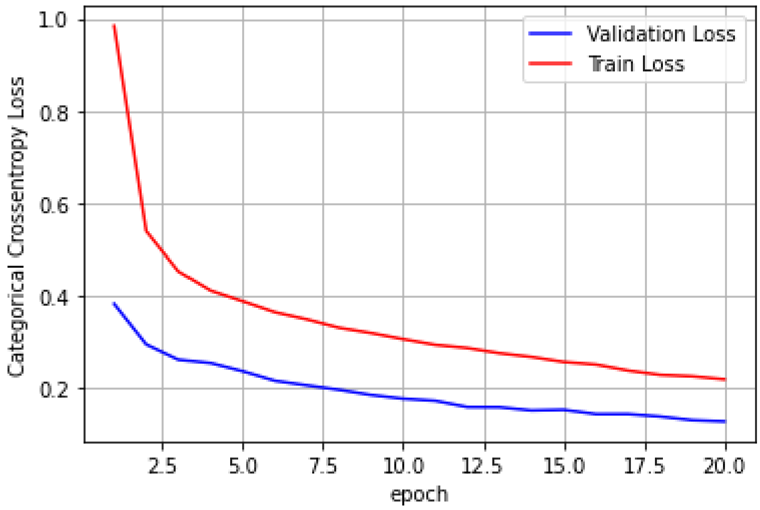
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accurac
y'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verb
ose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 9s 157us/step - loss: 0.983
8 - acc: 0.6901 - val_loss: 0.3823 - val_acc: 0.9008
Epoch 2/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.5394
- acc: 0.8332 - val_loss: 0.2944 - val_acc: 0.9163
Epoch 3/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.4524
- acc: 0.8621 - val_loss: 0.2613 - val_acc: 0.9254
Epoch 4/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.4109
- acc: 0.8760 - val_loss: 0.2539 - val_acc: 0.9267
Epoch 5/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.3879
- acc: 0.8828 - val_loss: 0.2362 - val_acc: 0.9329
Epoch 6/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.3642
- acc: 0.8904 - val_loss: 0.2155 - val_acc: 0.9381
Epoch 7/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.3487
- acc: 0.8970 - val_loss: 0.2056 - val_acc: 0.9408
Epoch 8/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.3305
- acc: 0.8998 - val_loss: 0.1960 - val_acc: 0.9451
Epoch 9/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.3190
- acc: 0.9041 - val_loss: 0.1848 - val_acc: 0.9496
Epoch 10/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.3058
- acc: 0.9087 - val_loss: 0.1767 - val_acc: 0.9499
Epoch 11/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.2935
- acc: 0.9121 - val_loss: 0.1722 - val_acc: 0.9513
Epoch 12/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2866
- acc: 0.9145 - val_loss: 0.1586 - val_acc: 0.9545
Epoch 13/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.2754
- acc: 0.9180 - val_loss: 0.1582 - val_acc: 0.9575
Epoch 14/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.2673
- acc: 0.9205 - val_loss: 0.1513 - val_acc: 0.9579
Epoch 15/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.2564
- acc: 0.9248 - val_loss: 0.1527 - val_acc: 0.9570
Epoch 16/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.2507
- acc: 0.9265 - val_loss: 0.1433 - val_acc: 0.9616
Epoch 17/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.2377
- acc: 0.9298 - val_loss: 0.1433 - val_acc: 0.9630
Epoch 18/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2283
- acc: 0.9319 - val_loss: 0.1380 - val_acc: 0.9638
Epoch 19/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.2253
- acc: 0.9349 - val_loss: 0.1301 - val_acc: 0.9639
Epoch 20/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.2188
- acc: 0.9347 - val_loss: 0.1274 - val_acc: 0.9651
```

In [78]:

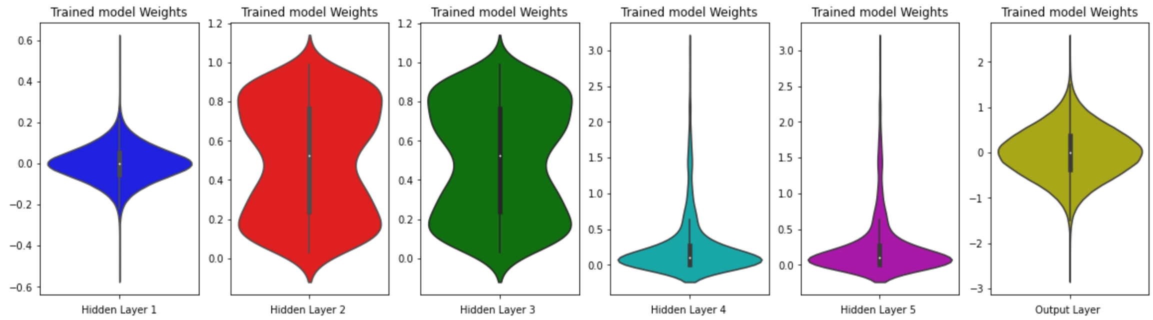
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.12741360827535392
Test accuracy: 0.9651
```



In [79]:

```
w_after = model_drop.get_weights()

h1_w = w_after[0].flatten().reshape(-1,1)
h2_w = w_after[4].flatten().reshape(-1,1)
h3_w = w_after[4].flatten().reshape(-1,1)
h3_w = w_after[4].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(figsize=(20,5))
plt.title("Weight matrices after model trained")
plt.subplot(1, 6, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h1_w,color='b')
plt.xlabel('Hidden Layer 1')

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

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

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

plt.subplot(1, 6, 5)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h5_w, color='m')
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()
```



Hyper-parameter tuning of Keras models using Sklearn

In [0]:

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

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


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

    return model
```

In [0]:

```
# https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-p
ython-keras/

activ = ['sigmoid','relu']

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

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

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

grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs =-1)
grid_result = grid.fit(X_train, Y_train)
```

In [82]:

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

```
Best: 0.977383 using {'activ': 'sigmoid'}
0.977383 (0.001965) with: {'activ': 'sigmoid'}
0.976550 (0.003953) with: {'activ': 'relu'}
```

In [98]:

```
# http://zetcode.com/python/prettytable/
from prettytable import PrettyTable

ptable = PrettyTable()

ptable.field_names = [ "Model", "Accuracy"]
ptable.add_row(["softmax",90.87])
ptable.add_row(["MLP+Sigmoid+SGDoptimizer",87.71])
ptable.add_row(["MLP+Sigmoid+ADAM",98.06])
ptable.add_row(["MLP+SGD+ReLU",96.27])
ptable.add_row(["MLP+ReLU+ADAM",97.6])
print(ptable)

ptable = PrettyTable()
ptable.field_names =["Model","Layers","Accuracy"]
ptable.add_row(["MLP+Batch-norm+ADAM",2,97.41])
ptable.add_row(["MLP+Batch-norm+ADAM",3,97.36])
ptable.add_row(["MLP+Batch-norm+ADAM",5,97.66])
ptable.add_row(["MLP+Batch-norm+ADAM",7,97.6])
print(ptable)

ptable = PrettyTable()
ptable.field_names= ["Model","dropouts","Accuracy"]
ptable.add_row(["MLP+dropouts+ADAM",2,97.89])
ptable.add_row(["MLP+Dropouts+ADAM",3,97.47])
ptable.add_row(["MLP+Dropouts+adam",5,97.4])
print(ptable)

ptable = PrettyTable()
ptable.field_names= ["Model","dropouts","layers","Accuracy"]
ptable.add_row(["MLP+Batch-norm+ADAM+DROPOUTS",2,2,96.7])
ptable.add_row(["MLP+Batch-norm+ADAM+DROPOUTS",7,7,97.42])
ptable.add_row([" MLP+Batch-norm+ADAM", 3,3,97.26 ])
ptable.add_row([" MLP+Batch-norm+ADAM+DROPOUTS", 5,5,96.51])
print(ptable)
```

Model	Accuracy
softmax	90.87
MLP+Sigmoid+SGDoptimizer	87.71
MLP+Sigmoid+ADAM	98.06
MLP+SGD+ReLU	96.27
MLP+ReLU+ADAM	97.6

Model	Layers	Accuracy
MLP+Batch-norm+ADAM	2	97.41
MLP+Batch-norm+ADAM	3	97.36
MLP+Batch-norm+ADAM	5	97.66
MLP+Batch-norm+ADAM	7	97.6

Model	dropouts	Accuracy
MLP+dropouts+ADAM	2	97.89
MLP+Dropouts+ADAM	3	97.47
MLP+Dropouts+adam	5	97.4

Model	dropouts	layers	Accuracy
MLP+Batch-norm+ADAM+DROPOUTS	2	2	96.7
MLP+Batch-norm+ADAM+DROPOUTS	7	7	97.42
MLP+Batch-norm+ADAM	3	3	97.26
MLP+Batch-norm+ADAM+DROPOUTS	5	5	96.51