Keras -- MLPs on MNIST

In [1]:

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

```
Using TensorFlow backend.
```

In [2]:

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

In [3]:

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

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 [5]:

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

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

In [6]:

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

print("Number of training examples :", X_train.shape[0], "and each image is of 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 [8]:

```
# if we observe the above matrix each cell is having a value between 0-255
# before we move to apply machine learning algorithms lets try to normalize the 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 [11]:

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

# The Sequential model is a linear stack of layers.
# you can create a Sequential model by passing a list of layer instances to the 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 [12]:

```
# some model parameters

output_dim = 10
input_dim = X_train.shape[1]

batch_size = 128
nb_epoch = 20
```

In [13]:

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

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

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

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

In [15]:

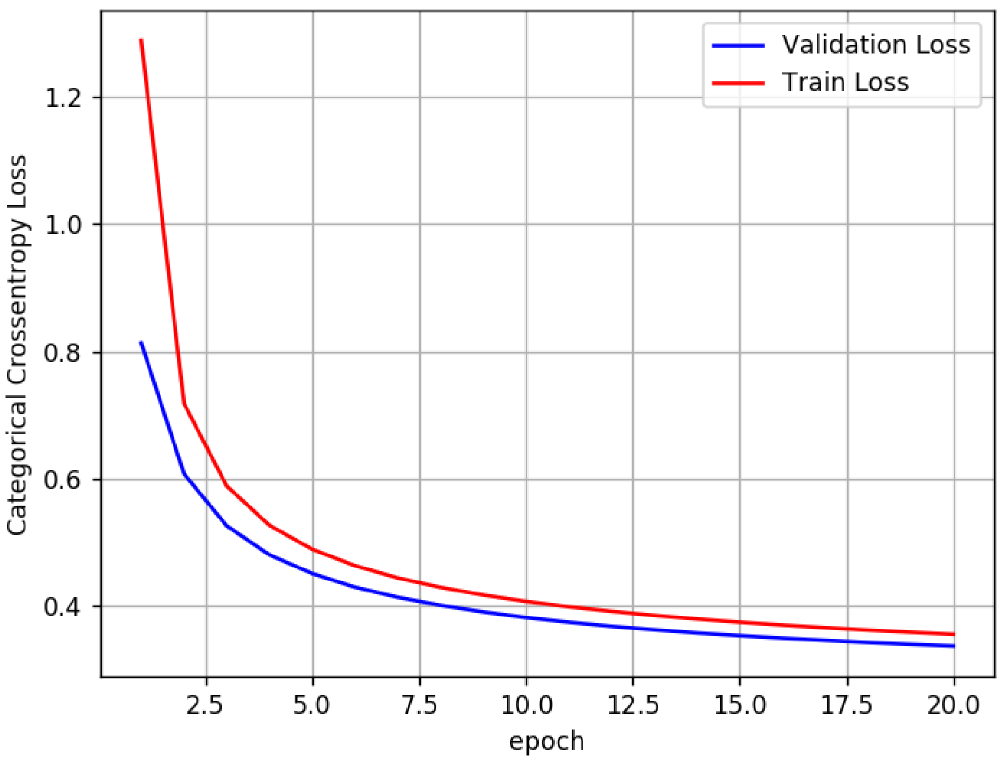
```
# Before training a model, you need to configure the learning process, which is done vi
a the compile method

# 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))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s 45us/step - loss: 1.2891
- accuracy: 0.6943 - val_loss: 0.8131 - val_accuracy: 0.8324
Epoch 2/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.7177
- accuracy: 0.8414 - val_loss: 0.6071 - val_accuracy: 0.8615
Epoch 3/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.5879
- accuracy: 0.8604 - val_loss: 0.5251 - val_accuracy: 0.8721
Epoch 4/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.5259
- accuracy: 0.8699 - val_loss: 0.4795 - val_accuracy: 0.8793
Epoch 5/20
60000/60000 [==============================] - 2s 32us/step - loss: 0.4883
- accuracy: 0.8757 - val_loss: 0.4502 - val_accuracy: 0.8842
Epoch 6/20
60000/60000 [==============================] - 2s 33us/step - loss: 0.4624
- accuracy: 0.8801 - val_loss: 0.4286 - val_accuracy: 0.8883
Epoch 7/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.4433
- accuracy: 0.8835 - val_loss: 0.4128 - val_accuracy: 0.8920
Epoch 8/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.4283
- accuracy: 0.8866 - val_loss: 0.4001 - val_accuracy: 0.8948
Epoch 9/20
60000/60000 [==============================] - 3s 46us/step - loss: 0.4163
- accuracy: 0.8885 - val_loss: 0.3896 - val_accuracy: 0.8975
Epoch 10/20
60000/60000 [==============================] - 2s 42us/step - loss: 0.4063
- accuracy: 0.8908 - val_loss: 0.3808 - val_accuracy: 0.8992
Epoch 11/20
60000/60000 [==============================] - 2s 42us/step - loss: 0.3978
- accuracy: 0.8928 - val_loss: 0.3735 - val_accuracy: 0.9007
Epoch 12/20
60000/60000 [==============================] - 3s 43us/step - loss: 0.3906
- accuracy: 0.8941 - val_loss: 0.3671 - val_accuracy: 0.9023
Epoch 13/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.3841
- accuracy: 0.8957 - val_loss: 0.3616 - val_accuracy: 0.9036
Epoch 14/20
60000/60000 [==============================] - 3s 47us/step - loss: 0.3785
- accuracy: 0.8968 - val_loss: 0.3566 - val_accuracy: 0.9048
Epoch 15/20
60000/60000 [==============================] - 3s 52us/step - loss: 0.3735
- accuracy: 0.8975 - val_loss: 0.3523 - val_accuracy: 0.9065
Epoch 16/20
60000/60000 [==============================] - 4s 63us/step - loss: 0.3690
- accuracy: 0.8988 - val_loss: 0.3484 - val_accuracy: 0.9066
Epoch 17/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.3649
- accuracy: 0.8996 - val_loss: 0.3449 - val_accuracy: 0.9073
Epoch 18/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.3611
- accuracy: 0.9005 - val_loss: 0.3415 - val_accuracy: 0.9086
Epoch 19/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.3577
- accuracy: 0.9015 - val_loss: 0.3386 - val_accuracy: 0.9082
Epoch 20/20
60000/60000 [==============================] - 3s 45us/step - loss: 0.3545
- accuracy: 0.9022 - val_loss: 0.3360 - val_accuracy: 0.9089
```

In [16]:

```
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: 0.33595913439393044
Test accuracy: 0.9089000225067139
```



MLP + Sigmoid activation + SGDOptimizer

In [17]:

```
# 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 [18]:

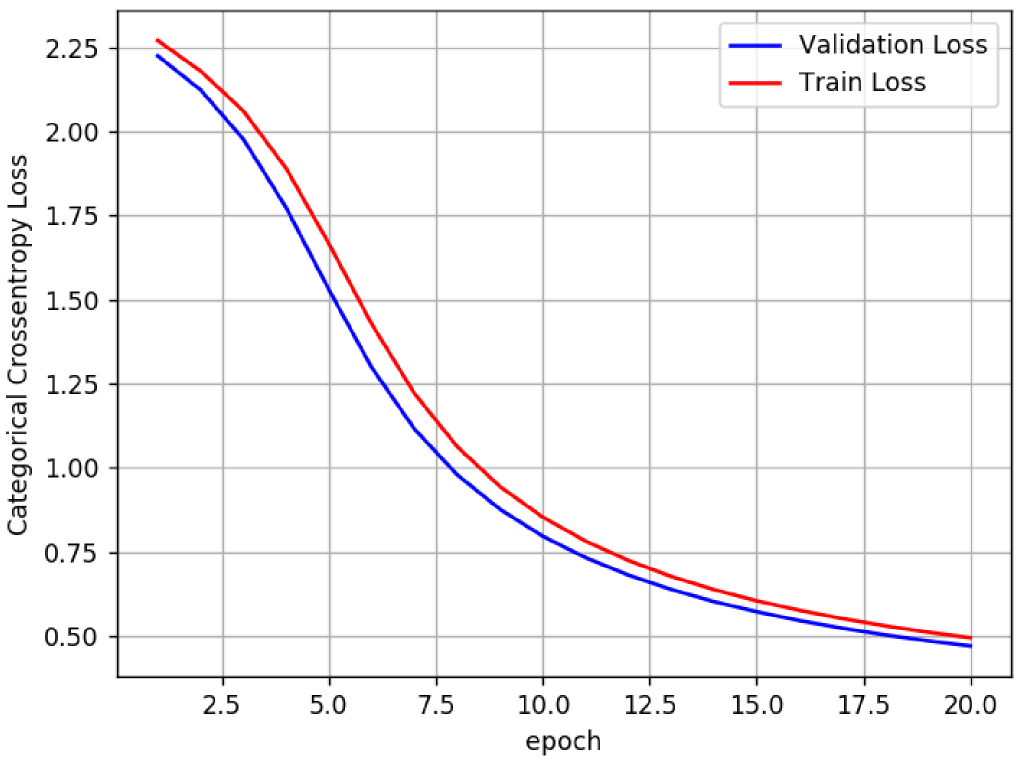
```
model_sigmoid.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accur
acy'])


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 [==============================] - 15s 244us/step - loss: 2.26
99 - accuracy: 0.2155 - val_loss: 2.2243 - val_accuracy: 0.3367
Epoch 2/20
60000/60000 [==============================] - 15s 250us/step - loss: 2.17
93 - accuracy: 0.4564 - val_loss: 2.1231 - val_accuracy: 0.4867
Epoch 3/20
60000/60000 [==============================] - 15s 253us/step - loss: 2.06
06 - accuracy: 0.5884 - val_loss: 1.9769 - val_accuracy: 0.6208
Epoch 4/20
60000/60000 [==============================] - 14s 225us/step - loss: 1.88
99 - accuracy: 0.6447 - val_loss: 1.7730 - val_accuracy: 0.6918
Epoch 5/20
60000/60000 [==============================] - 13s 216us/step - loss: 1.66
60 - accuracy: 0.6815 - val_loss: 1.5293 - val_accuracy: 0.7316
Epoch 6/20
60000/60000 [==============================] - 13s 223us/step - loss: 1.42
72 - accuracy: 0.7132 - val_loss: 1.2979 - val_accuracy: 0.7497
Epoch 7/20
60000/60000 [==============================] - 14s 225us/step - loss: 1.22
06 - accuracy: 0.7416 - val_loss: 1.1150 - val_accuracy: 0.7724
Epoch 8/20
60000/60000 [==============================] - 16s 267us/step - loss: 1.06
24 - accuracy: 0.7653 - val_loss: 0.9787 - val_accuracy: 0.7866
Epoch 9/20
60000/60000 [==============================] - 14s 225us/step - loss: 0.94
38 - accuracy: 0.7834 - val_loss: 0.8764 - val_accuracy: 0.7985
Epoch 10/20
60000/60000 [==============================] - 15s 249us/step - loss: 0.85
30 - accuracy: 0.7976 - val_loss: 0.7961 - val_accuracy: 0.8093
Epoch 11/20
60000/60000 [==============================] - 12s 207us/step - loss: 0.78
16 - accuracy: 0.8097 - val_loss: 0.7324 - val_accuracy: 0.8203
Epoch 12/20
60000/60000 [==============================] - 15s 247us/step - loss: 0.72
40 - accuracy: 0.8201 - val_loss: 0.6808 - val_accuracy: 0.8292
Epoch 13/20
60000/60000 [==============================] - 14s 234us/step - loss: 0.67
67 - accuracy: 0.8287 - val_loss: 0.6380 - val_accuracy: 0.8380
Epoch 14/20
60000/60000 [==============================] - 14s 237us/step - loss: 0.63
74 - accuracy: 0.8363 - val_loss: 0.6018 - val_accuracy: 0.8461
Epoch 15/20
60000/60000 [==============================] - 15s 250us/step - loss: 0.60
41 - accuracy: 0.8439 - val_loss: 0.5717 - val_accuracy: 0.8519
Epoch 16/20
60000/60000 [==============================] - 14s 238us/step - loss: 0.57
58 - accuracy: 0.8503 - val_loss: 0.5452 - val_accuracy: 0.8571
Epoch 17/20
60000/60000 [==============================] - 13s 215us/step - loss: 0.55
12 - accuracy: 0.8554 - val_loss: 0.5227 - val_accuracy: 0.8649
Epoch 18/20
60000/60000 [==============================] - 14s 239us/step - loss: 0.52
98 - accuracy: 0.8609 - val_loss: 0.5027 - val_accuracy: 0.8681
Epoch 19/20
60000/60000 [==============================] - 13s 218us/step - loss: 0.51
08 - accuracy: 0.8650 - val_loss: 0.4850 - val_accuracy: 0.8714
Epoch 20/20
60000/60000 [==============================] - 15s 247us/step - loss: 0.49
42 - accuracy: 0.8692 - val_loss: 0.4695 - val_accuracy: 0.8762
```

In [19]:

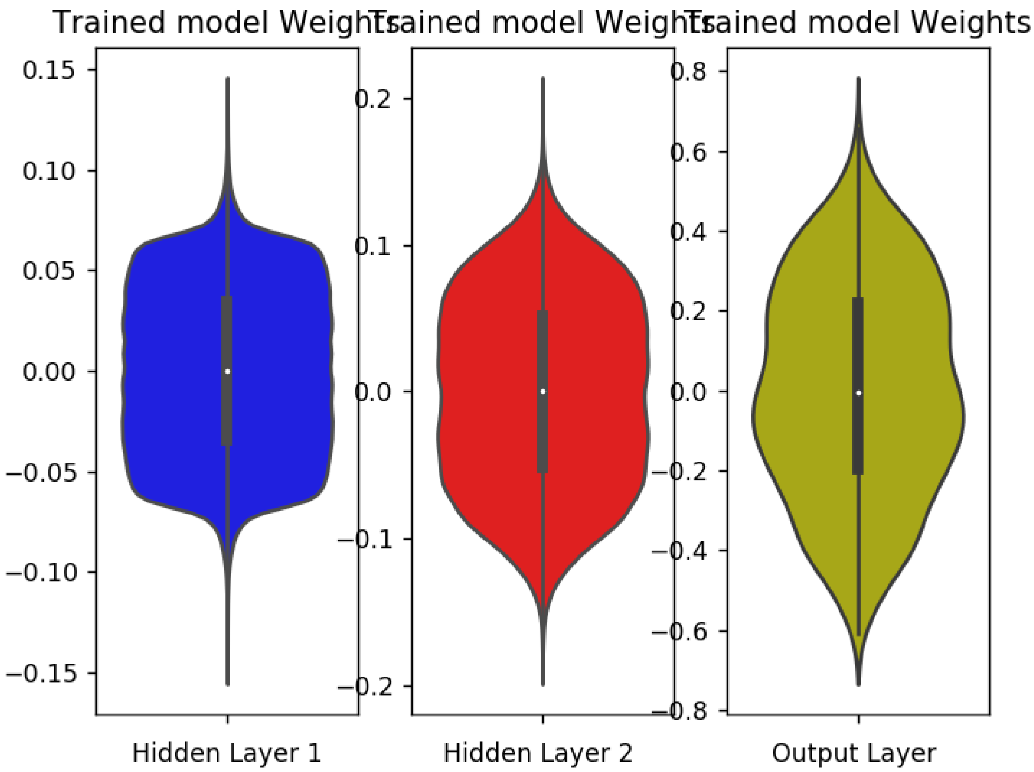
```
score = model_sigmoid.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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: 0.46953374400490723
Test accuracy: 0.8762000203132629
```



In [20]:

```
w_after = model_sigmoid.get_weights()

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


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

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

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



MLP + Sigmoid activation + ADAM

In [ ]:

```
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))
```

In [ ]:

```
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)
```

In [32]:

```
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 [33]:

```
# 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_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'))

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 [34]:

```
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 40us/step - loss: 0.7508
- acc: 0.7941 - val_loss: 0.3916 - val_acc: 0.8959
Epoch 2/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.3573
- acc: 0.9000 - val_loss: 0.3031 - val_acc: 0.9159
Epoch 3/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.2923
- acc: 0.9172 - val_loss: 0.2613 - val_acc: 0.9273
Epoch 4/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.2571
- acc: 0.9273 - val_loss: 0.2366 - val_acc: 0.9344
Epoch 5/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.2325
- acc: 0.9344 - val_loss: 0.2184 - val_acc: 0.9387
Epoch 6/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.2137
- acc: 0.9398 - val_loss: 0.2057 - val_acc: 0.9406
Epoch 7/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1986
- acc: 0.9437 - val_loss: 0.1926 - val_acc: 0.9462
Epoch 8/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1858
- acc: 0.9474 - val_loss: 0.1812 - val_acc: 0.9472
Epoch 9/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1747
- acc: 0.9506 - val_loss: 0.1759 - val_acc: 0.9492
Epoch 10/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.1651
- acc: 0.9536 - val_loss: 0.1646 - val_acc: 0.9511
Epoch 11/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.1566
- acc: 0.9559 - val_loss: 0.1588 - val_acc: 0.9527
Epoch 12/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1489
- acc: 0.9582 - val_loss: 0.1521 - val_acc: 0.9535
Epoch 13/20
60000/60000 [==============================] - 2s 34us/step - loss: 0.1421
- acc: 0.9604 - val_loss: 0.1472 - val_acc: 0.9553
Epoch 14/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.1358
- acc: 0.9625 - val_loss: 0.1447 - val_acc: 0.9575
Epoch 15/20
60000/60000 [==============================] - 2s 35us/step - loss: 0.1304
- acc: 0.9640 - val_loss: 0.1389 - val_acc: 0.9578
Epoch 16/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.1252
- acc: 0.9658 - val_loss: 0.1336 - val_acc: 0.9592
Epoch 17/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.1204
- acc: 0.9668 - val_loss: 0.1312 - val_acc: 0.9607
Epoch 18/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.1157
- acc: 0.9679 - val_loss: 0.1273 - val_acc: 0.9609
Epoch 19/20
60000/60000 [==============================] - 2s 36us/step - loss: 0.1117
- acc: 0.9693 - val_loss: 0.1242 - val_acc: 0.9623
Epoch 20/20
60000/60000 [==============================] - 2s 37us/step - loss: 0.1077
- acc: 0.9702 - val_loss: 0.1218 - val_acc: 0.9620
```

In [35]:

```
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: 0.12176313624158501
Test accuracy: 0.962
```

In [36]:

```
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 [37]:

```
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 [==============================] - 3s 46us/step - loss: 0.2240
- acc: 0.9326 - val_loss: 0.1131 - val_acc: 0.9679
Epoch 2/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0854
- acc: 0.9740 - val_loss: 0.0898 - val_acc: 0.9737
Epoch 3/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0521
- acc: 0.9842 - val_loss: 0.0764 - val_acc: 0.9760
Epoch 4/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0359
- acc: 0.9886 - val_loss: 0.0726 - val_acc: 0.9787
Epoch 5/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0278
- acc: 0.9911 - val_loss: 0.0807 - val_acc: 0.9757
Epoch 6/20
60000/60000 [==============================] - 3s 42us/step - loss: 0.0199
- acc: 0.9937 - val_loss: 0.0834 - val_acc: 0.9771
Epoch 7/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0176
- acc: 0.9947 - val_loss: 0.0843 - val_acc: 0.9772
Epoch 8/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0164
- acc: 0.9946 - val_loss: 0.0779 - val_acc: 0.9784
Epoch 9/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0117
- acc: 0.9963 - val_loss: 0.0768 - val_acc: 0.9799
Epoch 10/20
60000/60000 [==============================] - 3s 43us/step - loss: 0.0123
- acc: 0.9961 - val_loss: 0.0896 - val_acc: 0.9776
Epoch 11/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0137
- acc: 0.9954 - val_loss: 0.0857 - val_acc: 0.9786
Epoch 12/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0106
- acc: 0.9965 - val_loss: 0.0854 - val_acc: 0.9798
Epoch 13/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0081
- acc: 0.9974 - val_loss: 0.0976 - val_acc: 0.9775
Epoch 14/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0134
- acc: 0.9956 - val_loss: 0.0950 - val_acc: 0.9782
Epoch 15/20
60000/60000 [==============================] - 3s 43us/step - loss: 0.0076
- acc: 0.9977 - val_loss: 0.0835 - val_acc: 0.9810
```

```
Epoch 16/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0057
- acc: 0.9983 - val_loss: 0.1038 - val_acc: 0.9789
Epoch 17/20
60000/60000 [==============================] - 2s 42us/step - loss: 0.0115
- acc: 0.9960 - val_loss: 0.0916 - val_acc: 0.9799
Epoch 18/20
60000/60000 [==============================] - 2s 39us/step - loss: 0.0061
- acc: 0.9981 - val_loss: 0.0853 - val_acc: 0.9821
Epoch 19/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0076
- acc: 0.9976 - val_loss: 0.1079 - val_acc: 0.9767
Epoch 20/20
60000/60000 [==============================] - 2s 42us/step - loss: 0.0111
- acc: 0.9965 - val_loss: 0.0984 - val_acc: 0.9803
```

In [38]:

```
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: 0.09837867890719448
Test accuracy: 0.9803
```

In [39]:

```
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 [15]:

```
# 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()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_3 (Dense)              (None, 512)               401920    
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048      
_________________________________________________________________
dense_4 (Dense)              (None, 128)               65664     
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512       
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [41]:

```
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 81us/step - loss: 0.3037
- acc: 0.9101 - val_loss: 0.2233 - val_acc: 0.9354
Epoch 2/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.1737
- acc: 0.9496 - val_loss: 0.1665 - val_acc: 0.9505
Epoch 3/20
60000/60000 [==============================] - 4s 66us/step - loss: 0.1356
- acc: 0.9601 - val_loss: 0.1488 - val_acc: 0.9546
Epoch 4/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.1128
- acc: 0.9662 - val_loss: 0.1408 - val_acc: 0.9591
Epoch 5/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0959
- acc: 0.9713 - val_loss: 0.1229 - val_acc: 0.9624
Epoch 6/20
60000/60000 [==============================] - 4s 67us/step - loss: 0.0821
- acc: 0.9747 - val_loss: 0.1190 - val_acc: 0.9634
Epoch 7/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0681
- acc: 0.9781 - val_loss: 0.1126 - val_acc: 0.9652
Epoch 8/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0599
- acc: 0.9819 - val_loss: 0.1055 - val_acc: 0.9681
Epoch 9/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0520
- acc: 0.9833 - val_loss: 0.1084 - val_acc: 0.9668
Epoch 10/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0456
- acc: 0.9852 - val_loss: 0.1095 - val_acc: 0.9677
Epoch 11/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0389
- acc: 0.9874 - val_loss: 0.0959 - val_acc: 0.9724
Epoch 12/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0347
- acc: 0.9888 - val_loss: 0.0942 - val_acc: 0.9727
Epoch 13/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0308
- acc: 0.9899 - val_loss: 0.0996 - val_acc: 0.9712
Epoch 14/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0297
- acc: 0.9900 - val_loss: 0.1011 - val_acc: 0.9702
Epoch 15/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.0253
- acc: 0.9919 - val_loss: 0.0966 - val_acc: 0.9746
Epoch 16/20
60000/60000 [==============================] - 4s 68us/step - loss: 0.0202
- acc: 0.9937 - val_loss: 0.0874 - val_acc: 0.9751
Epoch 17/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.0216
- acc: 0.9930 - val_loss: 0.0954 - val_acc: 0.9723
Epoch 18/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.0199
- acc: 0.9935 - val_loss: 0.0985 - val_acc: 0.9723
Epoch 19/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0182
- acc: 0.9943 - val_loss: 0.0912 - val_acc: 0.9743
Epoch 20/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.0182
- acc: 0.9937 - val_loss: 0.0998 - val_acc: 0.9710
```

In [0]:

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

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

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

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_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: 0.09344355644605822
Test accuracy: 0.9757
```

In [0]:

```
w_after = model_batch.get_weights()

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


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

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

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

5. MLP + Dropout + AdamOptimizer

In [42]:

```
# 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()
```

```
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_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_17 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_18 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_4 (Batch (None, 128)               512       
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_19 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [43]:

```
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 86us/step - loss: 0.6731
- acc: 0.7920 - val_loss: 0.2850 - val_acc: 0.9165
Epoch 2/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.4320
- acc: 0.8679 - val_loss: 0.2531 - val_acc: 0.9277
Epoch 3/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.3834
- acc: 0.8831 - val_loss: 0.2284 - val_acc: 0.9334
Epoch 4/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.3546
- acc: 0.8919 - val_loss: 0.2173 - val_acc: 0.9347
Epoch 5/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.3393
- acc: 0.8976 - val_loss: 0.2106 - val_acc: 0.9384
Epoch 6/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.3192
- acc: 0.9044 - val_loss: 0.1996 - val_acc: 0.9391
Epoch 7/20
60000/60000 [==============================] - 4s 69us/step - loss: 0.3063
- acc: 0.9067 - val_loss: 0.1914 - val_acc: 0.9434
Epoch 8/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.2941
- acc: 0.9108 - val_loss: 0.1823 - val_acc: 0.9464
Epoch 9/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.2825
- acc: 0.9151 - val_loss: 0.1755 - val_acc: 0.9470
Epoch 10/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.2682
- acc: 0.9200 - val_loss: 0.1653 - val_acc: 0.9512
Epoch 11/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.2560
- acc: 0.9234 - val_loss: 0.1593 - val_acc: 0.9518
Epoch 12/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.2457
- acc: 0.9259 - val_loss: 0.1532 - val_acc: 0.9545
Epoch 13/20
60000/60000 [==============================] - 4s 72us/step - loss: 0.2367
- acc: 0.9290 - val_loss: 0.1488 - val_acc: 0.9557
Epoch 14/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.2264
- acc: 0.9314 - val_loss: 0.1350 - val_acc: 0.9593
Epoch 15/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.2182
- acc: 0.9328 - val_loss: 0.1305 - val_acc: 0.9624
Epoch 16/20
60000/60000 [==============================] - 4s 74us/step - loss: 0.2073
- acc: 0.9382 - val_loss: 0.1286 - val_acc: 0.9619
Epoch 17/20
60000/60000 [==============================] - 4s 70us/step - loss: 0.1986
- acc: 0.9389 - val_loss: 0.1192 - val_acc: 0.9661
Epoch 18/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.1907
- acc: 0.9424 - val_loss: 0.1152 - val_acc: 0.9663
Epoch 19/20
60000/60000 [==============================] - 4s 73us/step - loss: 0.1820
- acc: 0.9459 - val_loss: 0.1188 - val_acc: 0.9663
Epoch 20/20
60000/60000 [==============================] - 4s 71us/step - loss: 0.1789
- acc: 0.9463 - val_loss: 0.1063 - val_acc: 0.9693
```

In [44]:

```
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: 0.10633013175930828
Test accuracy: 0.9693
```

In [45]:

```
w_after = model_drop.get_weights()

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


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

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

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

Hyper-parameter tuning of Keras models using Sklearn

In [0]:

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

    model = Sequential()
    model.add(Dense(512, activation=activ, input_shape=(input_dim,), kernel_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 [47]:

```
# 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)
grid_result = grid.fit(X_train, Y_train)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:1
978: FutureWarning: The default value of cv will change from 3 to 5 in ver
sion 0.22. Specify it explicitly to silence this warning.
  warnings.warn(CV_WARNING, FutureWarning)
```

In [48]:

```
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: 0.975333 using {'activ': 'sigmoid'}
0.975333 (0.001211) with: {'activ': 'sigmoid'}
0.975083 (0.001109) with: {'activ': 'relu'}
```

In [0]:

```

```

ASSIGNMENT

** 2 LAYER ARCHITECTURE WITH BATCH NORMALIZATION AND DROPOUT**

In [16]:

```
from keras.layers import Dropout

model_2= Sequential()

model_2.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_2.add(BatchNormalization())
model_2.add(Dropout(0.5))

model_2.add(Dense(256, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )
model_2.add(BatchNormalization())
model_2.add(Dropout(0.5))

model_2.add(Dense(output_dim, activation='softmax'))


model_2.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_6 (Dense)              (None, 364)               285740    
_________________________________________________________________
batch_normalization_3 (Batch (None, 364)               1456      
_________________________________________________________________
dropout_1 (Dropout)          (None, 364)               0         
_________________________________________________________________
dense_7 (Dense)              (None, 256)               93440     
_________________________________________________________________
batch_normalization_4 (Batch (None, 256)               1024      
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_8 (Dense)              (None, 10)                2570      
=================================================================
Total params: 384,230
Trainable params: 382,990
Non-trainable params: 1,240
_________________________________________________________________
```

In [ ]:

```
model_2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'
])

history = model_2.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose
=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.491
8 - accuracy: 0.8500 - val_loss: 0.1702 - val_accuracy: 0.9486
Epoch 2/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.253
5 - accuracy: 0.9239 - val_loss: 0.1308 - val_accuracy: 0.9586
Epoch 3/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.206
6 - accuracy: 0.9378 - val_loss: 0.1118 - val_accuracy: 0.9644
Epoch 4/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.170
4 - accuracy: 0.9480 - val_loss: 0.0958 - val_accuracy: 0.9698
Epoch 5/20
30976/60000 [==============>...............] - ETA: 3s - loss: 0.1586 - ac
curacy: 0.9510
```

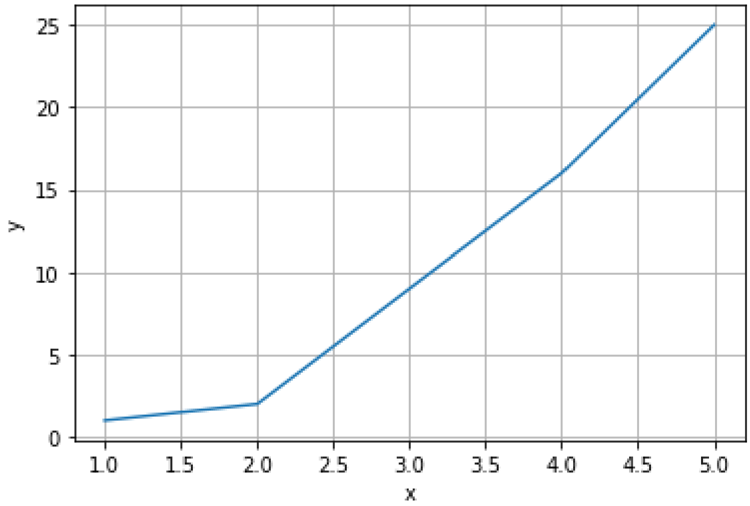
In [51]:

```
score = model_2.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: 0.054869781568923644
Test accuracy: 0.9822
```

In [1]:

```
import matplotlib.pyplot as plt
x=[1,2,3,4,5]
y=[1,2,9,16,25]
plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```



In [53]:

```
w_after = model_2.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()
```

** 2 LAYER ARCHITECTURE WITHOUT BATCH NORMALIZATION AND DROPOUT**

In [14]:

```
from keras.layers import Dropout

model_3= Sequential()

model_3.add(Dense(364, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))

model_3.add(Dense(256, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )

model_3.add(Dense(output_dim, activation='softmax'))


model_3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_2 (Dense)              (None, 364)               285740    
_________________________________________________________________
dense_3 (Dense)              (None, 256)               93440     
_________________________________________________________________
dense_4 (Dense)              (None, 10)                2570      
=================================================================
Total params: 381,750
Trainable params: 381,750
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

```
model_3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'
])

history = model_3.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose
=1, validation_data=(X_test, Y_test))
```

In [56]:

```
score = model_3.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: 0.11713699789448274
Test accuracy: 0.9784
```

In [57]:

```
w_after = model_2.get_weights()

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


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

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

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

3 LAYER ARCHITECTURE

WITH BATCH NORMALIZATION AND DROPOUT

In [59]:

```
from keras.layers import Dropout

model_4= Sequential()

model_4.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_4.add(BatchNormalization())
model_4.add(Dropout(0.2))

model_4.add(Dense(356, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )
model_4.add(BatchNormalization())
model_4.add(Dropout(0.2))

model_4.add(Dense(256, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )
model_4.add(BatchNormalization())
model_4.add(Dropout(0.2))


model_4.add(Dense(output_dim, activation='softmax'))


model_4.summary()
```

```
Model: "sequential_18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_50 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_9 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_7 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_51 (Dense)             (None, 356)               182628    
_________________________________________________________________
batch_normalization_10 (Batc (None, 356)               1424      
_________________________________________________________________
dropout_8 (Dropout)          (None, 356)               0         
_________________________________________________________________
dense_52 (Dense)             (None, 256)               91392     
_________________________________________________________________
batch_normalization_11 (Batc (None, 256)               1024      
_________________________________________________________________
dropout_9 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_53 (Dense)             (None, 10)                2570      
=================================================================
Total params: 683,006
Trainable params: 680,758
Non-trainable params: 2,248
_________________________________________________________________
```

In [60]:

```
model_4.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'
])

history = model_4.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose
=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 8s 125us/step - loss: 0.318
2 - acc: 0.9031 - val_loss: 0.1210 - val_acc: 0.9598
Epoch 2/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.1461
- acc: 0.9549 - val_loss: 0.0939 - val_acc: 0.9704
Epoch 3/20
60000/60000 [==============================] - 5s 87us/step - loss: 0.1101
- acc: 0.9653 - val_loss: 0.0824 - val_acc: 0.9733
Epoch 4/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0921
- acc: 0.9712 - val_loss: 0.0862 - val_acc: 0.9721
Epoch 5/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0780
- acc: 0.9751 - val_loss: 0.0747 - val_acc: 0.9766
Epoch 6/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0712
- acc: 0.9777 - val_loss: 0.0662 - val_acc: 0.9784
Epoch 7/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0651
- acc: 0.9790 - val_loss: 0.0697 - val_acc: 0.9792
Epoch 8/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0552
- acc: 0.9818 - val_loss: 0.0711 - val_acc: 0.9789
Epoch 9/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0524
- acc: 0.9824 - val_loss: 0.0695 - val_acc: 0.9801
Epoch 10/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0502
- acc: 0.9834 - val_loss: 0.0670 - val_acc: 0.9820
Epoch 11/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0459
- acc: 0.9848 - val_loss: 0.0631 - val_acc: 0.9812
Epoch 12/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0443
- acc: 0.9855 - val_loss: 0.0694 - val_acc: 0.9789
Epoch 13/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0379
- acc: 0.9874 - val_loss: 0.0643 - val_acc: 0.9822
Epoch 14/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0354
- acc: 0.9879 - val_loss: 0.0655 - val_acc: 0.9828
Epoch 15/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0327
- acc: 0.9894 - val_loss: 0.0696 - val_acc: 0.9807
Epoch 16/20
60000/60000 [==============================] - 5s 88us/step - loss: 0.0337
- acc: 0.9889 - val_loss: 0.0637 - val_acc: 0.9838
Epoch 17/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0298
- acc: 0.9902 - val_loss: 0.0675 - val_acc: 0.9812
Epoch 18/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0323
- acc: 0.9892 - val_loss: 0.0684 - val_acc: 0.9817
Epoch 19/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0278
- acc: 0.9904 - val_loss: 0.0727 - val_acc: 0.9806
Epoch 20/20
60000/60000 [==============================] - 5s 89us/step - loss: 0.0264
- acc: 0.9911 - val_loss: 0.0653 - val_acc: 0.9826
```

In [61]:

```
score = model_4.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: 0.06527992642150567
Test accuracy: 0.9826
```

In [62]:

```
print(score)
```

```
[0.06527992642150567, 0.9826]
```

In [64]:

```
w_after = model_4.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='y')
plt.xlabel('Hidden Layer 3 ')

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

WITHOUT BATCH NORMALIZATION AND DROPOUT

In [69]:

```
from keras.layers import Dropout

model_5= Sequential()

model_5.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))

model_5.add(Dense(356, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )

model_5.add(Dense(256, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )

model_5.add(Dense(output_dim, activation='softmax'))


model_5.summary()
```

```
Model: "sequential_19"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_54 (Dense)             (None, 512)               401920    
_________________________________________________________________
dense_55 (Dense)             (None, 356)               182628    
_________________________________________________________________
dense_56 (Dense)             (None, 256)               91392     
_________________________________________________________________
dense_57 (Dense)             (None, 10)                2570      
=================================================================
Total params: 678,510
Trainable params: 678,510
Non-trainable params: 0
_________________________________________________________________
```

In [70]:

```
model_5.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'
])

history = model_5.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose
=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 5s 81us/step - loss: 3.6271
- acc: 0.7508 - val_loss: 2.2213 - val_acc: 0.8383
Epoch 2/20
60000/60000 [==============================] - 3s 48us/step - loss: 2.1185
- acc: 0.8489 - val_loss: 2.0017 - val_acc: 0.8566
Epoch 3/20
60000/60000 [==============================] - 3s 48us/step - loss: 1.9423
- acc: 0.8620 - val_loss: 1.8992 - val_acc: 0.8636
Epoch 4/20
60000/60000 [==============================] - 3s 50us/step - loss: 1.8517
- acc: 0.8703 - val_loss: 1.8494 - val_acc: 0.8682
Epoch 5/20
60000/60000 [==============================] - 3s 50us/step - loss: 1.8119
- acc: 0.8738 - val_loss: 1.8758 - val_acc: 0.8637
Epoch 6/20
60000/60000 [==============================] - 3s 51us/step - loss: 1.7877
- acc: 0.8766 - val_loss: 1.8339 - val_acc: 0.8695
Epoch 7/20
60000/60000 [==============================] - 3s 51us/step - loss: 0.7151
- acc: 0.9378 - val_loss: 0.2033 - val_acc: 0.9661
Epoch 8/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.1027
- acc: 0.9800 - val_loss: 0.2098 - val_acc: 0.9661
Epoch 9/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0742
- acc: 0.9838 - val_loss: 0.1661 - val_acc: 0.9709
Epoch 10/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0688
- acc: 0.9850 - val_loss: 0.1831 - val_acc: 0.9693
Epoch 11/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0610
- acc: 0.9862 - val_loss: 0.1754 - val_acc: 0.9681
Epoch 12/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0578
- acc: 0.9866 - val_loss: 0.1677 - val_acc: 0.9692
Epoch 13/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0531
- acc: 0.9873 - val_loss: 0.1552 - val_acc: 0.9723
Epoch 14/20
60000/60000 [==============================] - 3s 50us/step - loss: 0.0492
- acc: 0.9890 - val_loss: 0.1644 - val_acc: 0.9737
Epoch 15/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0581
- acc: 0.9869 - val_loss: 0.1597 - val_acc: 0.9724
Epoch 16/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0413
- acc: 0.9904 - val_loss: 0.1712 - val_acc: 0.9712
Epoch 17/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0421
- acc: 0.9904 - val_loss: 0.1707 - val_acc: 0.9725
Epoch 18/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0393
- acc: 0.9911 - val_loss: 0.1690 - val_acc: 0.9736
Epoch 19/20
60000/60000 [==============================] - 3s 48us/step - loss: 0.0393
- acc: 0.9910 - val_loss: 0.1581 - val_acc: 0.9739
Epoch 20/20
60000/60000 [==============================] - 3s 49us/step - loss: 0.0430
- acc: 0.9905 - val_loss: 0.1619 - val_acc: 0.9734
```

In [71]:

```
score = model_5.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: 0.16188472667328224
Test accuracy: 0.9734
```

In [72]:

```
w_after = model_5.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='y')
plt.xlabel('Hidden Layer 3 ')

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

5 LAYER ARCHITECTURE

WITH BATCH NORMALIZATION AND DROPOUT

In [17]:

```
from keras.layers import Dropout

model_6= Sequential()

model_6.add(Dense(656, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_6.add(BatchNormalization())
model_6.add(Dropout(0.2))

model_6.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_6.add(BatchNormalization())
model_6.add(Dropout(0.2))

model_6.add(Dense(356, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )
model_6.add(BatchNormalization())
model_6.add(Dropout(0.2))

model_6.add(Dense(256, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )
model_6.add(BatchNormalization())
model_6.add(Dropout(0.2))

model_6.add(Dense(128, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_6.add(BatchNormalization())
model_6.add(Dropout(0.2))


model_6.add(Dense(output_dim, activation='softmax'))


model_6.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_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_6 (Dense)              (None, 656)               514960    
_________________________________________________________________
batch_normalization_3 (Batch (None, 656)               2624      
_________________________________________________________________
dropout_1 (Dropout)          (None, 656)               0         
_________________________________________________________________
dense_7 (Dense)              (None, 512)               336384    
_________________________________________________________________
batch_normalization_4 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_8 (Dense)              (None, 356)               182628    
_________________________________________________________________
batch_normalization_5 (Batch (None, 356)               1424      
_________________________________________________________________
dropout_3 (Dropout)          (None, 356)               0         
_________________________________________________________________
dense_9 (Dense)              (None, 256)               91392     
_________________________________________________________________
batch_normalization_6 (Batch (None, 256)               1024      
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_10 (Dense)             (None, 128)               32896     
_________________________________________________________________
batch_normalization_7 (Batch (None, 128)               512       
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 10)                1290      
=================================================================
Total params: 1,167,182
Trainable params: 1,163,366
Non-trainable params: 3,816
_________________________________________________________________
```

In [18]:

```
model_6.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'
])

history = model_6.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 /usr/local/lib/python3.6/dist-packages/tensorflow_
core/python/ops/math_grad.py:1424: where (from tensorflow.python.ops.array
_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
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 [==============================] - 16s 271us/step - loss: 0.33
19 - acc: 0.8975 - val_loss: 0.1201 - val_acc: 0.9633
Epoch 2/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.1423
- acc: 0.9556 - val_loss: 0.0979 - val_acc: 0.9701
Epoch 3/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.1083
- acc: 0.9666 - val_loss: 0.0835 - val_acc: 0.9741
Epoch 4/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0906
- acc: 0.9718 - val_loss: 0.0749 - val_acc: 0.9785
Epoch 5/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.0808
- acc: 0.9755 - val_loss: 0.0732 - val_acc: 0.9795
```

```
Epoch 6/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0705
- acc: 0.9783 - val_loss: 0.0689 - val_acc: 0.9792
Epoch 7/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0650
- acc: 0.9795 - val_loss: 0.0713 - val_acc: 0.9809
Epoch 8/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.0602
- acc: 0.9809 - val_loss: 0.0642 - val_acc: 0.9820
Epoch 9/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0533
- acc: 0.9829 - val_loss: 0.0747 - val_acc: 0.9797
Epoch 10/20
60000/60000 [==============================] - 5s 90us/step - loss: 0.0462
- acc: 0.9848 - val_loss: 0.0677 - val_acc: 0.9811
Epoch 11/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0461
- acc: 0.9857 - val_loss: 0.0576 - val_acc: 0.9830
Epoch 12/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.0459
- acc: 0.9851 - val_loss: 0.0621 - val_acc: 0.9824
Epoch 13/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0405
- acc: 0.9872 - val_loss: 0.0643 - val_acc: 0.9821
Epoch 14/20
60000/60000 [==============================] - 5s 91us/step - loss: 0.0366
- acc: 0.9882 - val_loss: 0.0656 - val_acc: 0.9814
Epoch 15/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.0352
- acc: 0.9887 - val_loss: 0.0642 - val_acc: 0.9821
Epoch 16/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0355
- acc: 0.9884 - val_loss: 0.0647 - val_acc: 0.9838
Epoch 17/20
60000/60000 [==============================] - 6s 95us/step - loss: 0.0327
- acc: 0.9897 - val_loss: 0.0622 - val_acc: 0.9835
Epoch 18/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0313
- acc: 0.9894 - val_loss: 0.0642 - val_acc: 0.9832
Epoch 19/20
60000/60000 [==============================] - 6s 92us/step - loss: 0.0298
- acc: 0.9901 - val_loss: 0.0585 - val_acc: 0.9844
Epoch 20/20
60000/60000 [==============================] - 6s 93us/step - loss: 0.0260
- acc: 0.9917 - val_loss: 0.0605 - val_acc: 0.9848
```

In [19]:

```
score = model_6.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: 0.06047782948987442
Test accuracy: 0.9848
```

In [22]:

```
w_after = model_6.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 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=h3_w,color='y')
plt.xlabel('Hidden Layer 3 ')


plt.subplot(2, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='b')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(2, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 5 ')


plt.subplot(2,3,3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='g')
plt.xlabel('OUTPUT LAYER WEIGHTS ')
plt.show()
```

WITHOUT BATCH NORMALIZATION AND DROPOUT

In [23]:

```
from keras.layers import Dropout

model_7= Sequential()

model_7.add(Dense(656, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))


model_7.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))

model_7.add(Dense(356, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )


model_7.add(Dense(256, activation='relu', kernel_initializer=RandomNormal(mean=0.0, std
dev=0.55, seed=None)) )

model_7.add(Dense(128, activation='relu', input_shape=(input_dim,), kernel_initializer=
RandomNormal(mean=0.0, stddev=0.039, seed=None)))

model_7.add(Dense(output_dim, activation='softmax'))


model_7.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_12 (Dense)             (None, 656)               514960    
_________________________________________________________________
dense_13 (Dense)             (None, 512)               336384    
_________________________________________________________________
dense_14 (Dense)             (None, 356)               182628    
_________________________________________________________________
dense_15 (Dense)             (None, 256)               91392     
_________________________________________________________________
dense_16 (Dense)             (None, 128)               32896     
_________________________________________________________________
dense_17 (Dense)             (None, 10)                1290      
=================================================================
Total params: 1,159,550
Trainable params: 1,159,550
Non-trainable params: 0
_________________________________________________________________
```

In [24]:

```
model_7.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'
])

history = model_7.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose
=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 3s 54us/step - loss: 1.7796
- acc: 0.8432 - val_loss: 0.2184 - val_acc: 0.9336
Epoch 2/20
60000/60000 [==============================] - 3s 42us/step - loss: 0.1299
- acc: 0.9609 - val_loss: 0.1096 - val_acc: 0.9680
Epoch 3/20
60000/60000 [==============================] - 3s 43us/step - loss: 0.0793
- acc: 0.9756 - val_loss: 0.0978 - val_acc: 0.9703
Epoch 4/20
60000/60000 [==============================] - 3s 42us/step - loss: 0.0637
- acc: 0.9805 - val_loss: 0.0898 - val_acc: 0.9760
Epoch 5/20
60000/60000 [==============================] - 2s 42us/step - loss: 0.0517
- acc: 0.9836 - val_loss: 0.0916 - val_acc: 0.9742
Epoch 6/20
60000/60000 [==============================] - 3s 42us/step - loss: 0.0478
- acc: 0.9845 - val_loss: 0.0996 - val_acc: 0.9735
Epoch 7/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0430
- acc: 0.9868 - val_loss: 0.0860 - val_acc: 0.9774
Epoch 8/20
60000/60000 [==============================] - 3s 44us/step - loss: 0.0335
- acc: 0.9899 - val_loss: 0.1082 - val_acc: 0.9732
Epoch 9/20
60000/60000 [==============================] - 3s 43us/step - loss: 0.0359
- acc: 0.9887 - val_loss: 0.1383 - val_acc: 0.9680
Epoch 10/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0325
- acc: 0.9904 - val_loss: 0.1018 - val_acc: 0.9777
Epoch 11/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0308
- acc: 0.9905 - val_loss: 0.1037 - val_acc: 0.9743
Epoch 12/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0282
- acc: 0.9912 - val_loss: 0.0858 - val_acc: 0.9788
Epoch 13/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0249
- acc: 0.9924 - val_loss: 0.0953 - val_acc: 0.9792
Epoch 14/20
60000/60000 [==============================] - 2s 41us/step - loss: 0.0249
- acc: 0.9922 - val_loss: 0.1075 - val_acc: 0.9772
Epoch 15/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0199
- acc: 0.9943 - val_loss: 0.1078 - val_acc: 0.9783
Epoch 16/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0227
- acc: 0.9929 - val_loss: 0.1000 - val_acc: 0.9752
Epoch 17/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0214
- acc: 0.9939 - val_loss: 0.0980 - val_acc: 0.9798
Epoch 18/20
60000/60000 [==============================] - 3s 42us/step - loss: 0.0150
- acc: 0.9953 - val_loss: 0.1106 - val_acc: 0.9778
Epoch 19/20
60000/60000 [==============================] - 2s 40us/step - loss: 0.0193
- acc: 0.9950 - val_loss: 0.1143 - val_acc: 0.9781
Epoch 20/20
60000/60000 [==============================] - 2s 42us/step - loss: 0.0199
- acc: 0.9944 - val_loss: 0.1074 - val_acc: 0.9790
```

In [25]:

```
score = model_7.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: 0.10735029972919778
Test accuracy: 0.979
```

In [26]:

```
w_after = model_7.get_weights()

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


fig = plt.figure()
plt.title("Weight matrices after model trained")
plt.subplot(1, 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=h3_w,color='y')
plt.xlabel('Hidden Layer 3 ')


plt.subplot(2, 3, 1)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='b')
plt.xlabel('Hidden Layer 4 ')

plt.subplot(2, 3, 2)
plt.title("Trained model Weights")
ax = sns.violinplot(y=h2_w, color='r')
plt.xlabel('Hidden Layer 5 ')


plt.subplot(2,3,3)
plt.title("Trained model Weights")
ax = sns.violinplot(y=out_w,color='g')
plt.xlabel('OUTPUT LAYER WEIGHTS ')
plt.show()
```

CONCLUSION

In [27]:

```
from prettytable import PrettyTable
x=PrettyTable(['ACTIVATION','OPTIMIZER','NO OF HIDDEN LAYERS','BATCHNORM AND DROPOUT',
'TEST ACCURACY','TEST SCORE'])
x.add_row(['RELU','ADAM','2','YES','0.9822','0.0548'])
x.add_row(['RELU','ADAM','2','NO','0.9784','0.117136'])
x.add_row(['RELU','ADAM','3','YES','0.9826','0.06527'])
x.add_row(['RELU','ADAM','3','NO','0.9734','0.16188'])
x.add_row(['RELU','ADAM','5','YES','0.9848','0.060477'])
x.add_row(['RELU','ADAM','5','NO','0.979','0.1073'])

print(x.get_string(start=0,end=9))
```

```
+------------+-----------+---------------------+-----------------------+--
-------------+------------+
| ACTIVATION | OPTIMIZER | NO OF HIDDEN LAYERS | BATCHNORM AND DROPOUT | T
EST ACCURACY | TEST SCORE |
+------------+-----------+---------------------+-----------------------+--
-------------+------------+
|    RELU    |    ADAM   |          2          |          YES          |
0.9822     |   0.0548   |
|    RELU    |    ADAM   |          2          |           NO          |
0.9784     |  0.117136  |
|    RELU    |    ADAM   |          3          |          YES          |
0.9826     |  0.06527   |
|    RELU    |    ADAM   |          3          |           NO          |
0.9734     |  0.16188   |
|    RELU    |    ADAM   |          5          |          YES          |
0.9848     |  0.060477  |
|    RELU    |    ADAM   |          5          |           NO          |
0.979      |   0.1073   |
+------------+-----------+---------------------+-----------------------+--
-------------+------------+
```

In [0]:

In [0]:

In [0]: