Keras -- MLPs on MNIST

In [1]:
```
# if you keras is not using tensorflow as backend set "KERAS_BACKEND=tensorflow" use this command
from keras.utils import np_utils
from keras.datasets import mnist
import seaborn as sns
from keras.initializers import RandomNormal
```

```
Using TensorFlow backend.
```

In [2]:
```
%matplotlib inline
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
import time
# https://gist.github.com/greydanus/f6eee59eaf1d90fcb3b534a25362cea4
# https://stackoverflow.com/a/14434334
# this function is used to update the plots for each epoch and error
def plt_dynamic(x, vy, ty, ax, colors=['b']):
    ax.plot(x, vy, 'b', label="Validation Loss")
    ax.plot(x, ty, 'r', label="Train Loss")
    plt.legend()
    plt.grid()
    fig.canvas.draw()
```

In [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.sh
ape[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 imag
e 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  25
5
 247 127   0   0   0   0   0   0   0   0   0   0   0   0  30  36  94  15
4
 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   8
2
  82  56  39   0   0   0   0   0   0   0   0   0   0   0   0  18 219  25
3
 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  15
4
   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  24
1
 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  18
```

```
7
   0   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 25
3
 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 25
3
 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 19
5
  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   1
6
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0
   0   0   0   0   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
```

```python
# before we move to apply machine learning algorithms lets try to norma
lize the data
# X => (X - Xmin)/(Xmax-Xmin) = X/255

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

In [9]:

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

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.01176471 0.07058824 0.07058824 0.07058824
 0.49411765 0.53333333 0.68627451 0.10196078 0.65098039 1.
 0.96862745 0.49803922 0.         0.         0.         0.

 0.         0.         0.         0.         0.         0.
```

```
0.         0.         0.11764706 0.14117647 0.36862745 0.60392157
0.66666667 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
0.88235294 0.6745098  0.99215686 0.94901961 0.76470588 0.25098039
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.19215686
0.93333333 0.99215686 0.99215686 0.99215686 0.99215686 0.99215686
0.99215686 0.99215686 0.99215686 0.98431373 0.36470588 0.32156863
0.32156863 0.21960784 0.15294118 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.07058824 0.85882353 0.99215686
0.99215686 0.99215686 0.99215686 0.99215686 0.77647059 0.71372549
0.96862745 0.94509804 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.31372549 0.61176471 0.41960784 0.99215686
0.99215686 0.80392157 0.04313725 0.         0.16862745 0.60392157
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.05490196 0.00392157 0.60392157 0.99215686 0.35294118
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.54509804 0.99215686 0.74509804 0.00784314 0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.04313725
0.74509804 0.99215686 0.2745098  0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.1372549  0.94509804
0.88235294 0.62745098 0.42352941 0.00392157 0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.

0.         0.         0.         0.31764706 0.94117647 0.99215686
```

```
0.99215686 0.46666667 0.09803922 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.17647059 0.72941176 0.99215686 0.99215686
0.58823529 0.10588235 0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.0627451  0.36470588 0.98823529 0.99215686 0.73333333
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.97647059 0.99215686 0.97647059 0.25098039 0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.18039216 0.50980392 0.71764706 0.99215686
0.99215686 0.81176471 0.00784314 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.15294118 0.58039216
0.89803922 0.99215686 0.99215686 0.99215686 0.98039216 0.71372549
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.09411765 0.44705882 0.86666667 0.99215686 0.99215686 0.99215686
0.99215686 0.78823529 0.30588235 0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.09019608 0.25882353 0.83529412 0.99215686
0.99215686 0.99215686 0.99215686 0.77647059 0.31764706 0.00784314
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.         0.
0.         0.         0.         0.         0.07058824 0.67058824
0.85882353 0.99215686 0.99215686 0.99215686 0.99215686 0.76470588
0.31372549 0.03529412 0.         0.         0.         0.

0.         0.         0.         0.         0.         0.
```

```
 0.          0.          0.          0.          0.          0.
 0.21568627 0.6745098  0.88627451 0.99215686 0.99215686 0.99215686
 0.99215686 0.95686275 0.52156863 0.04313725 0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.53333333 0.99215686
 0.99215686 0.99215686 0.83137255 0.52941176 0.51764706 0.0627451
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.        ]
```

In [10]:

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

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

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

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

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

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 instance
s to the constructor:

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

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

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

###

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

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

# Dense implements the operation: output = activation(dot(input, kerne
l) + bias) where
# activation is the element-wise activation function passed as the acti
```

```
vation argument,
# kernel is a weights matrix created by the layer, and
# bias is a bias vector created by the layer (only applicable if use_bi
as 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 throug
h 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, soft
max


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 [14]:
```
# Before training a model, you need to configure the learning process,
 which is done via the compile method

# It receives three arguments:
# An optimizer. This could be the string identifier of an existing opti
mizer , https://keras.io/optimizers/
# A loss function. This is the objective that the model will try to min
imize., https://keras.io/losses/
# A list of metrics. For any classification problem you will want to se
t this to metrics=['accuracy'].  https://keras.io/metrics/


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

# that is why we converted out labels into vectors

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

# Keras models are trained on Numpy arrays of input data and labels.
```

```
# For training a model, you will typically use the  fit function

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

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

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

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

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

```
WARNING:tensorflow:From C:\Users\hemant\AnacondaNew\lib\site-packages\k
eras\backend\tensorflow_backend.py:422: The name tf.global_variables is
deprecated. Please use tf.compat.v1.global_variables instead.

Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 2s 33us/step - loss: 1.2
452 - accuracy: 0.7094 - val_loss: 0.8040 - val_accuracy: 0.8294
Epoch 2/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.7
110 - accuracy: 0.8390 - val_loss: 0.6065 - val_accuracy: 0.8605
Epoch 3/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.5
854 - accuracy: 0.8582 - val_loss: 0.5265 - val_accuracy: 0.8727
Epoch 4/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.5
246 - accuracy: 0.8688 - val_loss: 0.4809 - val_accuracy: 0.8796
Epoch 5/20
60000/60000 [==============================] - 2s 29us/step - loss: 0.4
```

```
874 - accuracy: 0.8757 - val_loss: 0.4512 - val_accuracy: 0.8832
Epoch 6/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.4
618 - accuracy: 0.8804 - val_loss: 0.4300 - val_accuracy: 0.8876
Epoch 7/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.4
427 - accuracy: 0.8835 - val_loss: 0.4137 - val_accuracy: 0.8911
Epoch 8/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.4
278 - accuracy: 0.8863 - val_loss: 0.4008 - val_accuracy: 0.8933
Epoch 9/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.4
159 - accuracy: 0.8885 - val_loss: 0.3903 - val_accuracy: 0.8961
Epoch 10/20
60000/60000 [==============================] - 2s 29us/step - loss: 0.4
058 - accuracy: 0.8903 - val_loss: 0.3818 - val_accuracy: 0.8976
Epoch 11/20
60000/60000 [==============================] - 2s 29us/step - loss: 0.3
974 - accuracy: 0.8924 - val_loss: 0.3744 - val_accuracy: 0.8994
Epoch 12/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3
902 - accuracy: 0.8941 - val_loss: 0.3679 - val_accuracy: 0.9008
Epoch 13/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3
838 - accuracy: 0.8954 - val_loss: 0.3623 - val_accuracy: 0.9020
Epoch 14/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.3
782 - accuracy: 0.8970 - val_loss: 0.3574 - val_accuracy: 0.9037
Epoch 15/20
60000/60000 [==============================] - ETA: 0s - loss: 0.3727 -
accuracy: 0.89 - 2s 26us/step - loss: 0.3732 - accuracy: 0.8981 - val_l
oss: 0.3530 - val_accuracy: 0.9047
Epoch 16/20
60000/60000 [==============================] - 2s 29us/step - loss: 0.3
687 - accuracy: 0.8993 - val_loss: 0.3488 - val_accuracy: 0.9060
Epoch 17/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.3
646 - accuracy: 0.9000 - val_loss: 0.3453 - val_accuracy: 0.9062
Epoch 18/20
60000/60000 [==============================] - 2s 27us/step - loss: 0.3
```

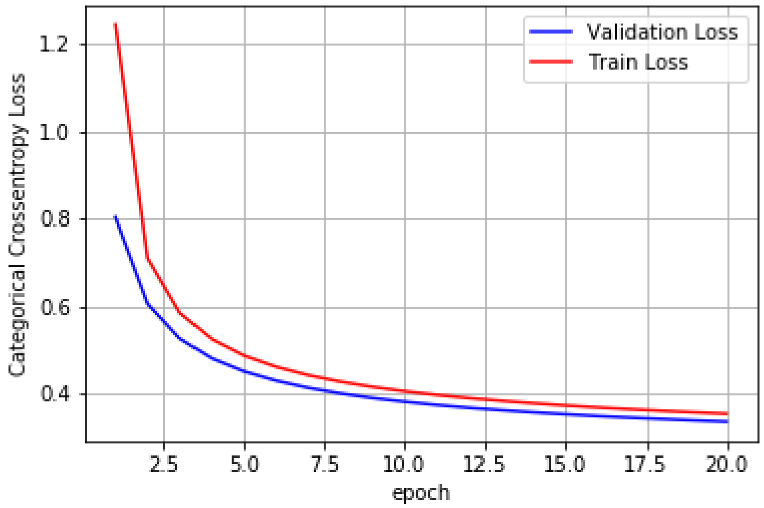
```
608 - accuracy: 0.9009 - val_loss: 0.3421 - val_accuracy: 0.9070

Epoch 19/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.3
574 - accuracy: 0.9016 - val_loss: 0.3389 - val_accuracy: 0.9082
Epoch 20/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3
543 - accuracy: 0.9025 - val_loss: 0.3361 - val_accuracy: 0.9089
```

In [15]:

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

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

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

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

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

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

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

```
Test score: 0.3360935353994369
```

Test accuracy: 0.9089000225067139



MLP + Sigmoid activation + SGDOptimizer

In [16]:
```
# Multilayer perceptron

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

model_sigmoid.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 512)               401920
_________________________________________________________________
```

```
dense_3 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

In [17]:
```
model_sigmoid.compile(optimizer='sgd', loss='categorical_crossentropy',
 metrics=['accuracy'])

history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, ep
ochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

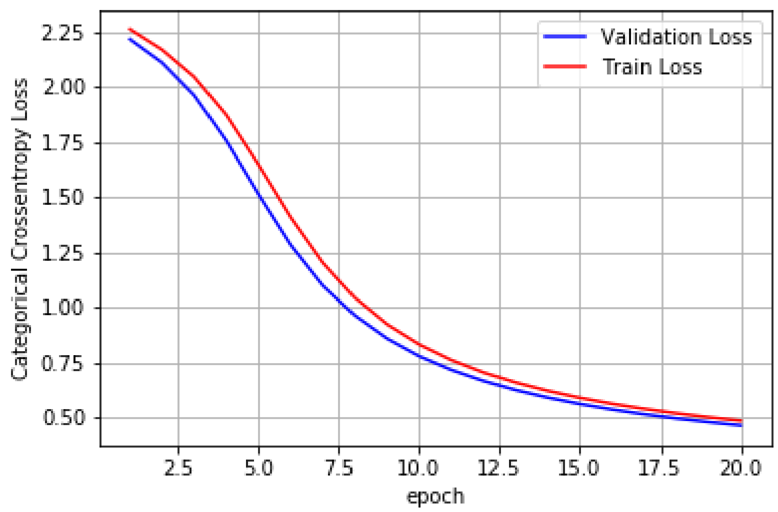
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 104us/step - loss: 2.
2607 - accuracy: 0.2370 - val_loss: 2.2153 - val_accuracy: 0.4839
Epoch 2/20
60000/60000 [==============================] - 6s 103us/step - loss: 2.
1682 - accuracy: 0.4685 - val_loss: 2.1094 - val_accuracy: 0.6104
Epoch 3/20
60000/60000 [==============================] - 6s 107us/step - loss: 2.
0450 - accuracy: 0.5841 - val_loss: 1.9610 - val_accuracy: 0.6890
Epoch 4/20
60000/60000 [==============================] - 6s 102us/step - loss: 1.
8702 - accuracy: 0.6382 - val_loss: 1.7538 - val_accuracy: 0.6820
Epoch 5/20
60000/60000 [==============================] - 6s 105us/step - loss: 1.
6456 - accuracy: 0.6795 - val_loss: 1.5117 - val_accuracy: 0.6971
Epoch 6/20
60000/60000 [==============================] - 6s 98us/step - loss: 1.4
083 - accuracy: 0.7151 - val_loss: 1.2827 - val_accuracy: 0.7655
Epoch 7/20
60000/60000 [==============================] - 6s 98us/step - loss: 1.2
021 - accuracy: 0.7509 - val_loss: 1.0988 - val_accuracy: 0.7740
Epoch 8/20
60000/60000 [==============================] - 6s 99us/step - loss: 1.0
```

```
422 - accuracy: 0.7746 - val_loss: 0.9620 - val_accuracy: 0.7872
Epoch 9/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.9
222 - accuracy: 0.7928 - val_loss: 0.8579 - val_accuracy: 0.8055
Epoch 10/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.8
312 - accuracy: 0.8073 - val_loss: 0.7779 - val_accuracy: 0.8194
Epoch 11/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.7
604 - accuracy: 0.8185 - val_loss: 0.7157 - val_accuracy: 0.8267
Epoch 12/20
60000/60000 [==============================] - 6s 98us/step - loss: 0.7
041 - accuracy: 0.8279 - val_loss: 0.6658 - val_accuracy: 0.8368
Epoch 13/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.
6582 - accuracy: 0.8360 - val_loss: 0.6239 - val_accuracy: 0.8444
Epoch 14/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.
6205 - accuracy: 0.8424 - val_loss: 0.5897 - val_accuracy: 0.8506
Epoch 15/20
60000/60000 [==============================] - 7s 121us/step - loss: 0.
5888 - accuracy: 0.8487 - val_loss: 0.5607 - val_accuracy: 0.8545
Epoch 16/20
60000/60000 [==============================] - 10s 168us/step - loss:
0.5618 - accuracy: 0.8540 - val_loss: 0.5358 - val_accuracy: 0.8606
Epoch 17/20
60000/60000 [==============================] - 7s 109us/step - loss: 0.
5386 - accuracy: 0.8591 - val_loss: 0.5140 - val_accuracy: 0.8643
Epoch 18/20
60000/60000 [==============================] - 8s 134us/step - loss: 0.
5183 - accuracy: 0.8637 - val_loss: 0.4951 - val_accuracy: 0.8673
Epoch 19/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.
5006 - accuracy: 0.8674 - val_loss: 0.4788 - val_accuracy: 0.8706
Epoch 20/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.
4849 - accuracy: 0.8708 - val_loss: 0.4638 - val_accuracy: 0.8759
```

In [18]:
```
score = model_sigmoid.evaluate(X_test, Y_test, verbose=0)
```

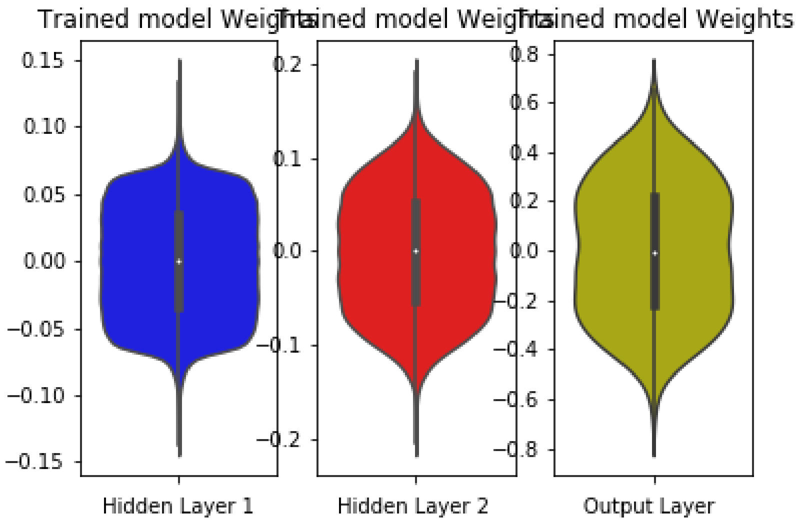
```
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.46379549462795255
Test accuracy: 0.8758999705314636
```



In [19]:
```
w_after = model_sigmoid.get_weights()

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


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

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

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



MLP + Sigmoid activation + ADAM

In [20]:
```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(512, activation='sigmoid', input_shape=(input_d
im,)))
model_sigmoid.add(Dense(128, activation='sigmoid'))
model_sigmoid.add(Dense(output_dim, activation='softmax'))

model_sigmoid.summary()

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

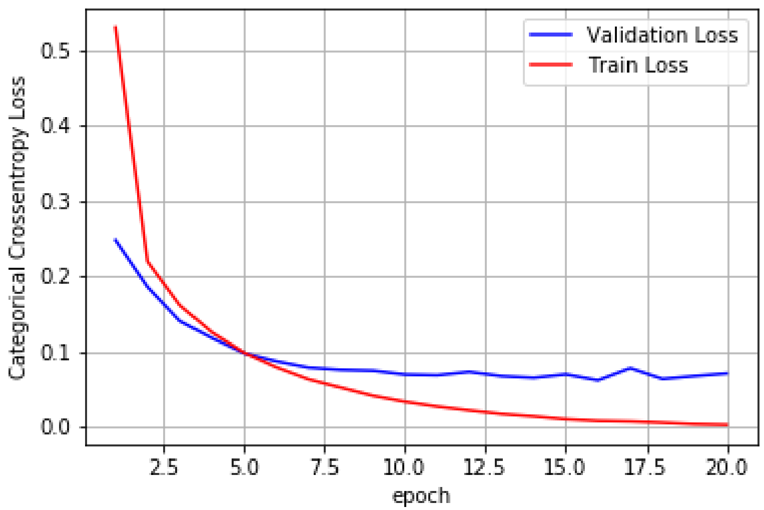
```
history = model_sigmoid.fit(X_train, Y_train, batch_size=batch_size, ep
ochs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_5 (Dense)              (None, 512)               401920    
_________________________________________________________________
dense_6 (Dense)              (None, 128)               65664     
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290      
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 10s 165us/step - loss:
0.5295 - accuracy: 0.8626 - val_loss: 0.2482 - val_accuracy: 0.9280
Epoch 2/20
60000/60000 [==============================] - 10s 168us/step - loss:
0.2197 - accuracy: 0.9359 - val_loss: 0.1859 - val_accuracy: 0.9446
Epoch 3/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.
1618 - accuracy: 0.9522 - val_loss: 0.1409 - val_accuracy: 0.9570
Epoch 4/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.
1257 - accuracy: 0.9628 - val_loss: 0.1183 - val_accuracy: 0.9650
Epoch 5/20
60000/60000 [==============================] - 8s 132us/step - loss: 0.
0990 - accuracy: 0.9714 - val_loss: 0.0983 - val_accuracy: 0.9697
Epoch 6/20
60000/60000 [==============================] - 8s 128us/step - loss: 0.
0799 - accuracy: 0.9765 - val_loss: 0.0878 - val_accuracy: 0.9724
Epoch 7/20
60000/60000 [==============================] - 8s 139us/step - loss: 0.
0640 - accuracy: 0.9813 - val_loss: 0.0794 - val_accuracy: 0.9745
Epoch 8/20
```

```
60000/60000 [==============================] - 8s 141us/step - loss: 0.
0529 - accuracy: 0.9844 - val_loss: 0.0764 - val_accuracy: 0.9769
Epoch 9/20
60000/60000 [==============================] - 11s 186us/step - loss:
0.0421 - accuracy: 0.9875 - val_loss: 0.0754 - val_accuracy: 0.9770
Epoch 10/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.
0341 - accuracy: 0.9905 - val_loss: 0.0703 - val_accuracy: 0.9788
Epoch 11/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.
0278 - accuracy: 0.9926 - val_loss: 0.0694 - val_accuracy: 0.9790
Epoch 12/20
60000/60000 [==============================] - 7s 122us/step - loss: 0.
0227 - accuracy: 0.9938 - val_loss: 0.0738 - val_accuracy: 0.9781
Epoch 13/20
60000/60000 [==============================] - 10s 159us/step - loss:
0.0179 - accuracy: 0.9959 - val_loss: 0.0681 - val_accuracy: 0.9793
Epoch 14/20
60000/60000 [==============================] - 9s 151us/step - loss: 0.
0147 - accuracy: 0.9962 - val_loss: 0.0660 - val_accuracy: 0.9813
Epoch 15/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.
0110 - accuracy: 0.9977 - val_loss: 0.0705 - val_accuracy: 0.9805
Epoch 16/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.
0088 - accuracy: 0.9983 - val_loss: 0.0626 - val_accuracy: 0.9822
Epoch 17/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.
0080 - accuracy: 0.9982 - val_loss: 0.0786 - val_accuracy: 0.9784
Epoch 18/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.
0064 - accuracy: 0.9987 - val_loss: 0.0644 - val_accuracy: 0.9831
Epoch 19/20
60000/60000 [==============================] - 7s 124us/step - loss: 0.
0045 - accuracy: 0.9991 - val_loss: 0.0683 - val_accuracy: 0.9817
Epoch 20/20
60000/60000 [==============================] - 7s 123us/step - loss: 0.
0036 - accuracy: 0.9994 - val_loss: 0.0717 - val_accuracy: 0.9798
```

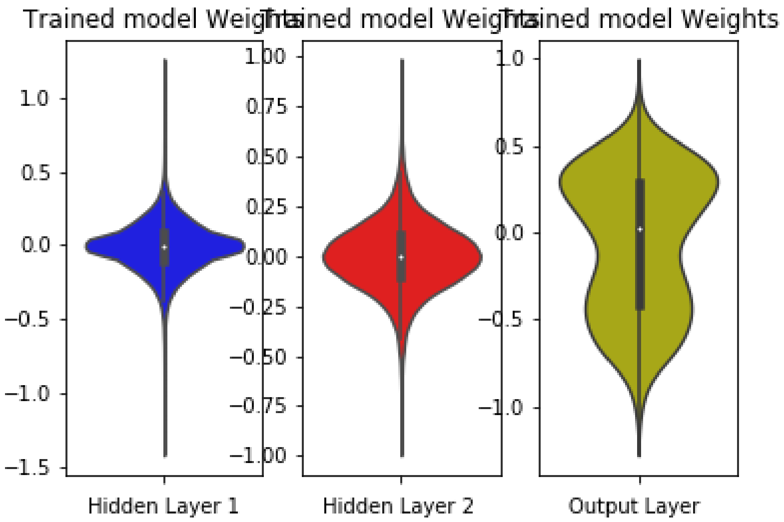
In [21]:
```
score = model_sigmoid.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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

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

```
Test score: 0.07166502268087498
Test accuracy: 0.9797999858856201
```



In [22]:
```
w_after = model_sigmoid.get_weights()

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


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

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

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



MLP + ReLU +SGD

In [23]:
```
# Multilayer perceptron

# https://arxiv.org/pdf/1707.09725.pdf#page=95
# for relu layers
# If we sample weights from a normal distribution N(0,σ) we satisfy thi
s condition with σ=√(2/(ni).
# h1 =>  σ=√(2/(fan_in) = 0.062  => N(0,σ) = N(0,0.062)
# h2 =>  σ=√(2/(fan_in)  = 0.125  => N(0,σ) = N(0,0.125)
# out =>  σ=√(2/(fan_in+1) = 0.120  => N(0,σ) = N(0,0.120)

model_relu = Sequential()
```

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

model_relu.summary()
```

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

In [24]:
```
model_relu.compile(optimizer='sgd', loss='categorical_crossentropy', me
trics=['accuracy'])

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epoch
s=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.
7175 - accuracy: 0.8036 - val_loss: 0.3799 - val_accuracy: 0.8941
Epoch 2/20
60000/60000 [==============================] - 6s 94us/step - loss: 0.3
428 - accuracy: 0.9044 - val_loss: 0.2972 - val_accuracy: 0.9151
Epoch 3/20
60000/60000 [==============================] - 6s 96us/step - loss: 0.2
830 - accuracy: 0.9199 - val_loss: 0.2570 - val_accuracy: 0.9264
Epoch 4/20
```

```
60000/60000 [==============================] - 6s 94us/step - loss: 0.2
498 - accuracy: 0.9293 - val_loss: 0.2357 - val_accuracy: 0.9321
Epoch 5/20
60000/60000 [==============================] - 6s 97us/step - loss: 0.2
267 - accuracy: 0.9354 - val_loss: 0.2170 - val_accuracy: 0.9369
Epoch 6/20
60000/60000 [==============================] - 7s 120us/step - loss: 0.
2087 - accuracy: 0.9410 - val_loss: 0.2025 - val_accuracy: 0.9423
Epoch 7/20
60000/60000 [==============================] - 10s 166us/step - loss:
0.1943 - accuracy: 0.9449 - val_loss: 0.1921 - val_accuracy: 0.9429
Epoch 8/20
60000/60000 [==============================] - 10s 171us/step - loss:
0.1821 - accuracy: 0.9492 - val_loss: 0.1818 - val_accuracy: 0.9459
Epoch 9/20
60000/60000 [==============================] - 9s 154us/step - loss: 0.
1716 - accuracy: 0.9514 - val_loss: 0.1739 - val_accuracy: 0.9489
Epoch 10/20
60000/60000 [==============================] - 9s 146us/step - loss: 0.
1623 - accuracy: 0.9545 - val_loss: 0.1664 - val_accuracy: 0.9496
Epoch 11/20
60000/60000 [==============================] - 9s 143us/step - loss: 0.
1544 - accuracy: 0.9568 - val_loss: 0.1600 - val_accuracy: 0.9517
Epoch 12/20
60000/60000 [==============================] - 9s 149us/step - loss: 0.
1472 - accuracy: 0.9590 - val_loss: 0.1544 - val_accuracy: 0.9525
Epoch 13/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.
1408 - accuracy: 0.9609 - val_loss: 0.1502 - val_accuracy: 0.9535
Epoch 14/20
60000/60000 [==============================] - 9s 142us/step - loss: 0.
1348 - accuracy: 0.9626 - val_loss: 0.1467 - val_accuracy: 0.9560
Epoch 15/20
60000/60000 [==============================] - 9s 149us/step - loss: 0.
1292 - accuracy: 0.9647 - val_loss: 0.1413 - val_accuracy: 0.9571
Epoch 16/20
60000/60000 [==============================] - 8s 141us/step - loss: 0.
1242 - accuracy: 0.9658 - val_loss: 0.1362 - val_accuracy: 0.9596
Epoch 17/20
```

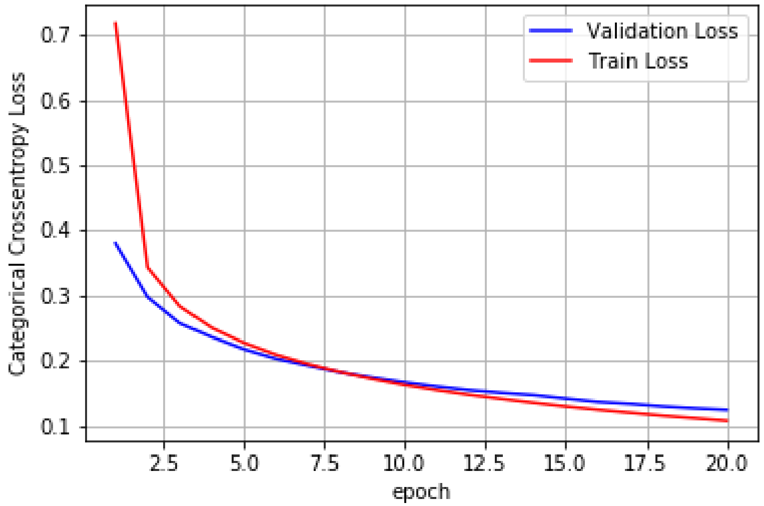
```
60000/60000 [==============================] - 9s 148us/step - loss: 0.
1196 - accuracy: 0.9671 - val_loss: 0.1332 - val_accuracy: 0.9589
Epoch 18/20
60000/60000 [==============================] - 8s 140us/step - loss: 0.
1151 - accuracy: 0.9688 - val_loss: 0.1293 - val_accuracy: 0.9605
Epoch 19/20
60000/60000 [==============================] - 9s 150us/step - loss: 0.
1113 - accuracy: 0.9695 - val_loss: 0.1264 - val_accuracy: 0.9613
Epoch 20/20
60000/60000 [==============================] - 8s 140us/step - loss: 0.
1074 - accuracy: 0.9703 - val_loss: 0.1240 - val_accuracy: 0.9620
```

In [25]:

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

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

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

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

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

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

vy = history.history['val_loss']
```

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

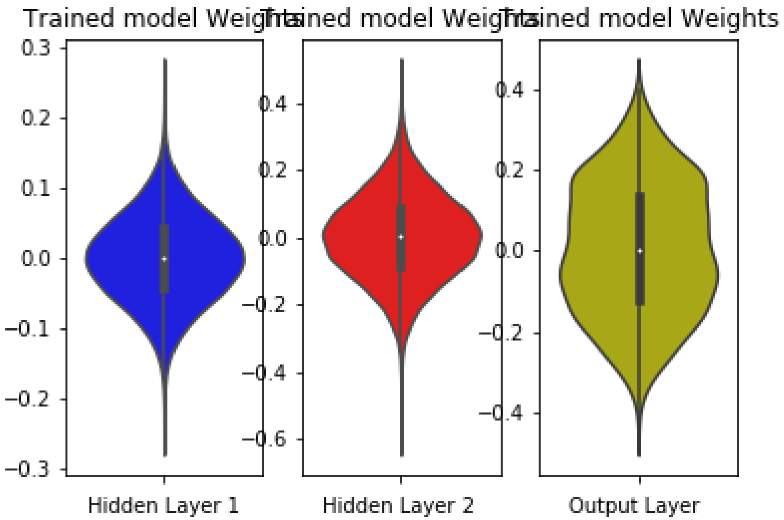
```
Test score: 0.12400117258317768
Test accuracy: 0.9620000123977661
```



In [26]:

```
w_after = model_relu.get_weights()

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


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

plt.subplot(1, 3, 2)
```

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

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



MLP + ReLU + ADAM

In [27]:
```
model_relu = Sequential()
model_relu.add(Dense(512, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.062, seed=None)))
model_relu.add(Dense(128, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.125, seed=None)) )
model_relu.add(Dense(output_dim, activation='softmax'))
```

```
print(model_relu.summary())

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

history = model_relu.fit(X_train, Y_train, batch_size=batch_size, epochs=nb_epoch, verbose=1, 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 [==============================] - 12s 198us/step - loss: 0.2276 - accuracy: 0.9326 - val_loss: 0.1111 - val_accuracy: 0.9658
Epoch 2/20
60000/60000 [==============================] - 10s 159us/step - loss: 0.0849 - accuracy: 0.9743 - val_loss: 0.0918 - val_accuracy: 0.9696
Epoch 3/20
60000/60000 [==============================] - 10s 160us/step - loss: 0.0529 - accuracy: 0.9841 - val_loss: 0.0718 - val_accuracy: 0.9776
Epoch 4/20
60000/60000 [==============================] - 10s 162us/step - loss: 0.0355 - accuracy: 0.9887 - val_loss: 0.0739 - val_accuracy: 0.9758
Epoch 5/20
60000/60000 [==============================] - 10s 165us/step - loss: 0.0272 - accuracy: 0.9913 - val_loss: 0.0785 - val_accuracy: 0.9762
Epoch 6/20
```

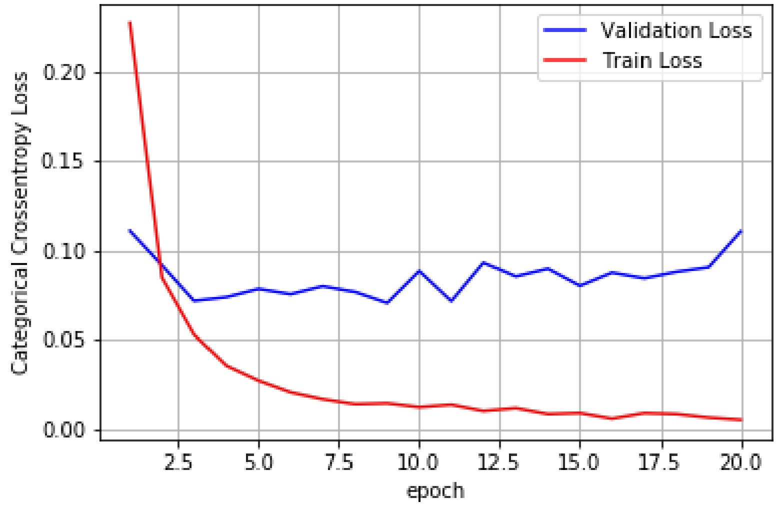
```
60000/60000 [==============================] - 10s 168us/step - loss: 

0.0206 - accuracy: 0.9936 - val_loss: 0.0755 - val_accuracy: 0.9791
Epoch 7/20
60000/60000 [==============================] - 10s 169us/step - loss: 
0.0168 - accuracy: 0.9945 - val_loss: 0.0800 - val_accuracy: 0.9775
Epoch 8/20
60000/60000 [==============================] - 10s 170us/step - loss: 
0.0140 - accuracy: 0.9951 - val_loss: 0.0768 - val_accuracy: 0.9796
Epoch 9/20
60000/60000 [==============================] - 10s 172us/step - loss: 
0.0144 - accuracy: 0.9952 - val_loss: 0.0706 - val_accuracy: 0.9818
Epoch 10/20
60000/60000 [==============================] - 11s 179us/step - loss: 
0.0123 - accuracy: 0.9957 - val_loss: 0.0886 - val_accuracy: 0.9785
Epoch 11/20
60000/60000 [==============================] - 11s 181us/step - loss: 
0.0136 - accuracy: 0.9955 - val_loss: 0.0716 - val_accuracy: 0.9809
Epoch 12/20
60000/60000 [==============================] - 10s 171us/step - loss: 
0.0101 - accuracy: 0.9966 - val_loss: 0.0932 - val_accuracy: 0.9772
Epoch 13/20
60000/60000 [==============================] - 10s 174us/step - loss: 
0.0117 - accuracy: 0.9962 - val_loss: 0.0855 - val_accuracy: 0.9802
Epoch 14/20
60000/60000 [==============================] - 10s 165us/step - loss: 
0.0084 - accuracy: 0.9972 - val_loss: 0.0899 - val_accuracy: 0.9790
Epoch 15/20
60000/60000 [==============================] - 10s 172us/step - loss: 
0.0089 - accuracy: 0.9971 - val_loss: 0.0803 - val_accuracy: 0.9805
Epoch 16/20
60000/60000 [==============================] - 11s 181us/step - loss: 
0.0058 - accuracy: 0.9980 - val_loss: 0.0877 - val_accuracy: 0.9799
Epoch 17/20
60000/60000 [==============================] - 10s 169us/step - loss: 
0.0089 - accuracy: 0.9972 - val_loss: 0.0845 - val_accuracy: 0.9826
Epoch 18/20
60000/60000 [==============================] - 11s 183us/step - loss: 
0.0084 - accuracy: 0.9974 - val_loss: 0.0881 - val_accuracy: 0.9811
Epoch 19/20
```

```
60000/60000 [==============================] - 14s 235us/step - loss: 

0.0064 - accuracy: 0.9978 - val_loss: 0.0907 - val_accuracy: 0.9818
Epoch 20/20
60000/60000 [==============================] - 14s 241us/step - loss: 
0.0052 - accuracy: 0.9983 - val_loss: 0.1108 - val_accuracy: 0.9772
```

In [28]:

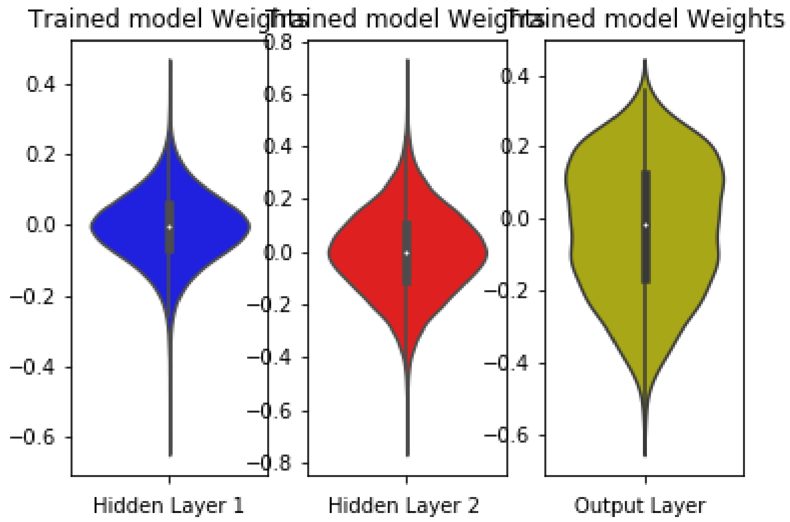
```
score = model_relu.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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

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

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

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

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


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

```
Test score: 0.11079006246657537
Test accuracy: 0.9771999716758728
```



In [29]:
```
w_after = model_relu.get_weights()

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


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

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

plt.subplot(1, 3, 3)
```

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



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

In [30]:
```
# Multilayer perceptron

# https://intoli.com/blog/neural-network-initialization/
# If we sample weights from a normal distribution N(0,σ) we satisfy thi
s condition with σ=√(2/(ni+ni+1).
# h1 =>  σ=√(2/(ni+ni+1) = 0.039  => N(0,σ) = N(0,0.039)
# h2 =>  σ=√(2/(ni+ni+1) = 0.055  => N(0,σ) = N(0,0.055)
# h1 =>  σ=√(2/(ni+ni+1) = 0.120  => N(0,σ) = N(0,0.120)

from keras.layers.normalization import BatchNormalization
```

```
model_batch = Sequential()

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

model_batch.add(Dense(128, activation='sigmoid', kernel_initializer=Ran
domNormal(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_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_14 (Dense)             (None, 512)               401920    
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048      
_________________________________________________________________
dense_15 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512       
_________________________________________________________________
dense_16 (Dense)             (None, 10)                1290      
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
_________________________________________________________________
```

In [31]:
```
model_batch.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

```
history = model_batch.fit(X_train, Y_train, batch_size=batch_size, epoc
hs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 21s 342us/step - loss:
0.3066 - accuracy: 0.9081 - val_loss: 0.2064 - val_accuracy: 0.9407
Epoch 2/20
60000/60000 [==============================] - 16s 259us/step - loss:
0.1801 - accuracy: 0.9462 - val_loss: 0.1706 - val_accuracy: 0.9512
Epoch 3/20
60000/60000 [==============================] - 18s 295us/step - loss:
0.1398 - accuracy: 0.9588 - val_loss: 0.1561 - val_accuracy: 0.9541
Epoch 4/20
60000/60000 [==============================] - 19s 320us/step - loss:
0.1130 - accuracy: 0.9661 - val_loss: 0.1399 - val_accuracy: 0.9596
Epoch 5/20
60000/60000 [==============================] - 18s 297us/step - loss:
0.0944 - accuracy: 0.9720 - val_loss: 0.1235 - val_accuracy: 0.9628
Epoch 6/20
60000/60000 [==============================] - 19s 320us/step - loss:
0.0808 - accuracy: 0.9750 - val_loss: 0.1175 - val_accuracy: 0.9650
Epoch 7/20
60000/60000 [==============================] - 19s 322us/step - loss:
0.0706 - accuracy: 0.9780 - val_loss: 0.1228 - val_accuracy: 0.9622
Epoch 8/20
60000/60000 [==============================] - 18s 293us/step - loss:
0.0584 - accuracy: 0.9818 - val_loss: 0.1043 - val_accuracy: 0.9697
Epoch 9/20
60000/60000 [==============================] - 18s 307us/step - loss:
0.0500 - accuracy: 0.9845 - val_loss: 0.1051 - val_accuracy: 0.9683
Epoch 10/20
60000/60000 [==============================] - 18s 305us/step - loss:
0.0462 - accuracy: 0.9854 - val_loss: 0.1049 - val_accuracy: 0.9701
Epoch 11/20
60000/60000 [==============================] - 18s 306us/step - loss:
0.0434 - accuracy: 0.9862 - val_loss: 0.1046 - val_accuracy: 0.9702
Epoch 12/20
60000/60000 [==============================] - 18s 297us/step - loss:
0.0346 - accuracy: 0.9890 - val_loss: 0.0970 - val_accuracy: 0.9719
```

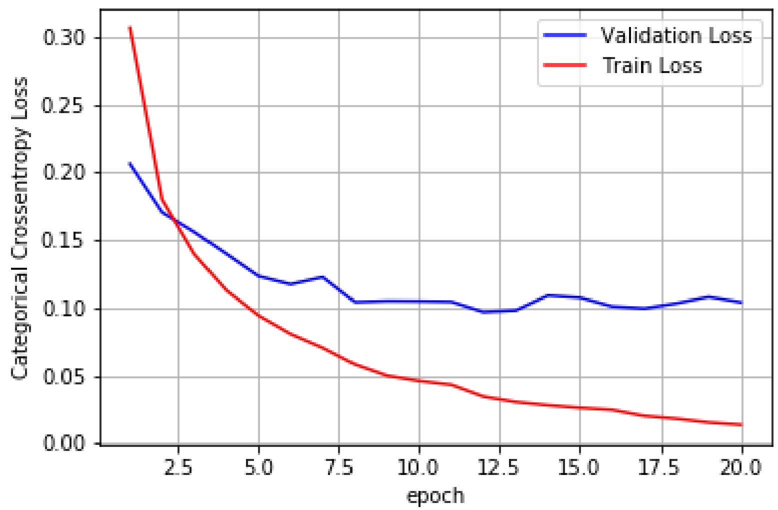
```
Epoch 13/20
60000/60000 [==============================] - 19s 309us/step - loss: 0.0306 - accuracy: 0.9901 - val_loss: 0.0982 - val_accuracy: 0.9723
Epoch 14/20
60000/60000 [==============================] - 18s 305us/step - loss: 0.0282 - accuracy: 0.9908 - val_loss: 0.1092 - val_accuracy: 0.9710
Epoch 15/20
60000/60000 [==============================] - 18s 303us/step - loss: 0.0263 - accuracy: 0.9916 - val_loss: 0.1076 - val_accuracy: 0.9700
Epoch 16/20
60000/60000 [==============================] - 18s 308us/step - loss: 0.0248 - accuracy: 0.9915 - val_loss: 0.1011 - val_accuracy: 0.9722
Epoch 17/20
60000/60000 [==============================] - 18s 305us/step - loss: 0.0203 - accuracy: 0.9934 - val_loss: 0.0997 - val_accuracy: 0.9740
Epoch 18/20
60000/60000 [==============================] - 18s 301us/step - loss: 0.0183 - accuracy: 0.9941 - val_loss: 0.1034 - val_accuracy: 0.9734
Epoch 19/20
60000/60000 [==============================] - 19s 316us/step - loss: 0.0155 - accuracy: 0.9948 - val_loss: 0.1081 - val_accuracy: 0.9721
Epoch 20/20
60000/60000 [==============================] - 18s 296us/step - loss: 0.0139 - accuracy: 0.9953 - val_loss: 0.1042 - val_accuracy: 0.9723
```

In [32]:

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

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

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

# print(history.history.keys())
# dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
# history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epo
```

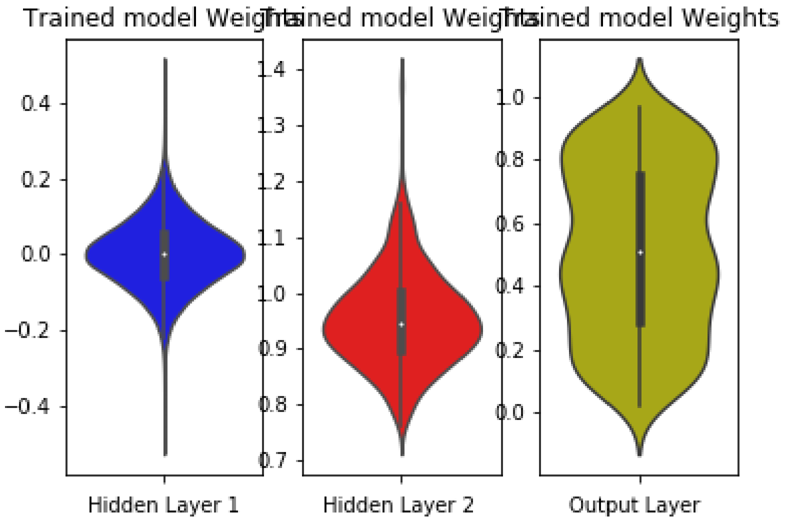
```
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.10415376693319994
Test accuracy: 0.9722999930381775
```



In [33]:
```
w_after = model_batch.get_weights()
```

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


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

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

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



5. MLP + Dropout + AdamOptimizer

In [34]:
```
# https://stackoverflow.com/questions/34716454/where-do-i-call-the-batc
hnormalization-function-in-keras

from keras.layers import Dropout

model_drop = Sequential()

model_drop.add(Dense(512, activation='sigmoid', input_shape=(input_dim
,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None
)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(128, activation='sigmoid', kernel_initializer=Rand
omNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
Model: "sequential_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 [35]:
```python
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

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

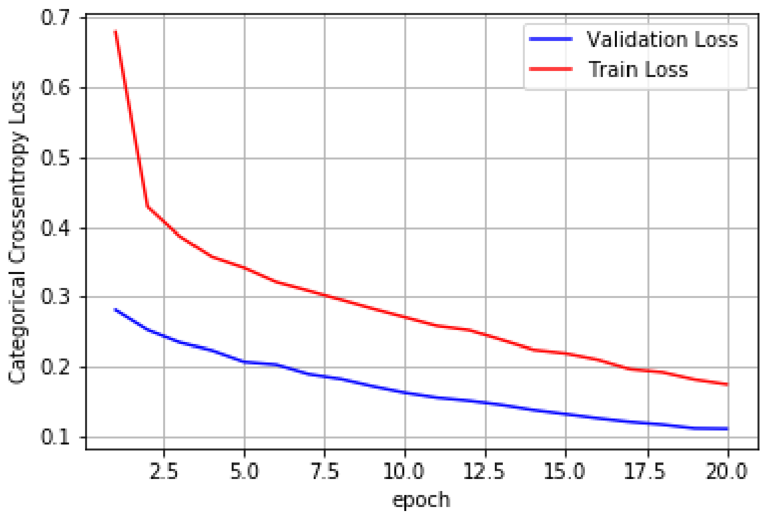
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 23s 383us/step - loss: 0.6794 - accuracy: 0.7914 - val_loss: 0.2815 - val_accuracy: 0.9163
Epoch 2/20
60000/60000 [==============================] - 20s 336us/step - loss: 0.4291 - accuracy: 0.8691 - val_loss: 0.2529 - val_accuracy: 0.9248
Epoch 3/20
60000/60000 [==============================] - 20s 325us/step - loss: 0.3868 - accuracy: 0.8824 - val_loss: 0.2349 - val_accuracy: 0.9311
Epoch 4/20
60000/60000 [==============================] - 19s 322us/step - loss: 0.3575 - accuracy: 0.8934 - val_loss: 0.2235 - val_accuracy: 0.9352
Epoch 5/20
60000/60000 [==============================] - 20s 339us/step - loss: 0.3419 - accuracy: 0.8969 - val_loss: 0.2072 - val_accuracy: 0.9394
Epoch 6/20
60000/60000 [==============================] - 19s 323us/step - loss: 0.3215 - accuracy: 0.9015 - val_loss: 0.2033 - val_accuracy: 0.9403
Epoch 7/20
60000/60000 [==============================] - 19s 319us/step - loss: 0.3090 - accuracy: 0.9073 - val_loss: 0.1898 - val_accuracy: 0.9429
```

```
Epoch 8/20
60000/60000 [==============================] - 20s 334us/step - loss:
0.2960 - accuracy: 0.9116 - val_loss: 0.1827 - val_accuracy: 0.9458
Epoch 9/20
60000/60000 [==============================] - 20s 340us/step - loss:
0.2830 - accuracy: 0.9154 - val_loss: 0.1722 - val_accuracy: 0.9476
Epoch 10/20
60000/60000 [==============================] - 20s 331us/step - loss:
0.2708 - accuracy: 0.9180 - val_loss: 0.1630 - val_accuracy: 0.9490
Epoch 11/20
60000/60000 [==============================] - 21s 342us/step - loss:
0.2584 - accuracy: 0.9219 - val_loss: 0.1559 - val_accuracy: 0.9534
Epoch 12/20
60000/60000 [==============================] - 20s 337us/step - loss:
0.2524 - accuracy: 0.9242 - val_loss: 0.1515 - val_accuracy: 0.9548
Epoch 13/20
60000/60000 [==============================] - 20s 339us/step - loss:
0.2385 - accuracy: 0.9294 - val_loss: 0.1455 - val_accuracy: 0.9572
Epoch 14/20
60000/60000 [==============================] - 21s 343us/step - loss:
0.2243 - accuracy: 0.9315 - val_loss: 0.1380 - val_accuracy: 0.9594
Epoch 15/20
60000/60000 [==============================] - 21s 356us/step - loss:
0.2193 - accuracy: 0.9345 - val_loss: 0.1323 - val_accuracy: 0.9602
Epoch 16/20
60000/60000 [==============================] - 20s 332us/step - loss:
0.2103 - accuracy: 0.9374 - val_loss: 0.1263 - val_accuracy: 0.9634
Epoch 17/20
60000/60000 [==============================] - 20s 330us/step - loss:
0.1968 - accuracy: 0.9405 - val_loss: 0.1210 - val_accuracy: 0.9634
Epoch 18/20
60000/60000 [==============================] - 20s 333us/step - loss:
0.1924 - accuracy: 0.9421 - val_loss: 0.1172 - val_accuracy: 0.9655
Epoch 19/20
60000/60000 [==============================] - 20s 341us/step - loss:
0.1820 - accuracy: 0.9458 - val_loss: 0.1118 - val_accuracy: 0.9666
Epoch 20/20
60000/60000 [==============================] - 20s 332us/step - loss:
0.1751 - accuracy: 0.9473 - val_loss: 0.1113 - val_accuracy: 0.9669
```

In [36]:

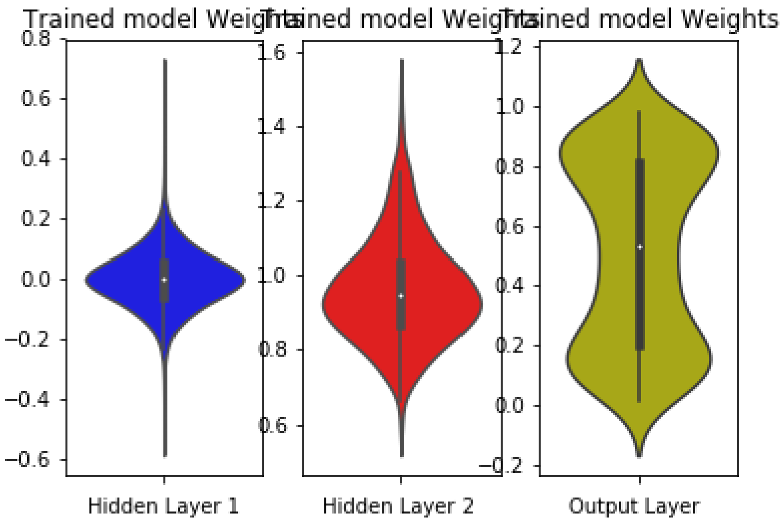
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.11127507402859628
Test accuracy: 0.9668999910354614
```



In [37]:
```
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 [38]:

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

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

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

    return model
```

In [39]:
```
# https://machinelearningmastery.com/grid-search-hyperparameters-deep-l
earning-models-python-keras/

activ = ['sigmoid','relu']

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

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

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

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

In [40]:
```
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: 0.977750 using {'activ': 'relu'}
0.977000 (0.001574) with: {'activ': 'sigmoid'}
0.977750 (0.001906) with: {'activ': 'relu'}
```

784-400-250-10 model with dropout

In [41]:

```
model_drop = Sequential()

model_drop.add(Dense(400, activation='relu', input_shape=(input_dim,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(250, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
Model: "sequential_19"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_53 (Dense)             (None, 400)               314000
_________________________________________________________________
dropout_3 (Dropout)          (None, 400)               0
_________________________________________________________________
dense_54 (Dense)             (None, 250)               100250
_________________________________________________________________
dropout_4 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_55 (Dense)             (None, 10)                2510
=================================================================
Total params: 416,760
Trainable params: 416,760
Non-trainable params: 0
_________________________________________________________________
```

In [42]:

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

```
history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epoch
s=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 19s 317us/step - loss:
0.7494 - accuracy: 0.7843 - val_loss: 0.2166 - val_accuracy: 0.9387
Epoch 2/20
60000/60000 [==============================] - 16s 269us/step - loss:
0.3349 - accuracy: 0.9019 - val_loss: 0.1584 - val_accuracy: 0.9544
Epoch 3/20
60000/60000 [==============================] - 17s 287us/step - loss:
0.2618 - accuracy: 0.9238 - val_loss: 0.1323 - val_accuracy: 0.9612
Epoch 4/20
60000/60000 [==============================] - 15s 245us/step - loss:
0.2214 - accuracy: 0.9358 - val_loss: 0.1222 - val_accuracy: 0.9640
Epoch 5/20
60000/60000 [==============================] - 16s 267us/step - loss:
0.2000 - accuracy: 0.9417 - val_loss: 0.1071 - val_accuracy: 0.9668
Epoch 6/20
60000/60000 [==============================] - 15s 257us/step - loss:
0.1816 - accuracy: 0.9475 - val_loss: 0.1076 - val_accuracy: 0.9698
Epoch 7/20
60000/60000 [==============================] - 17s 282us/step - loss:
0.1662 - accuracy: 0.9522 - val_loss: 0.0901 - val_accuracy: 0.9737
Epoch 8/20
60000/60000 [==============================] - 16s 272us/step - loss:
0.1595 - accuracy: 0.9533 - val_loss: 0.0920 - val_accuracy: 0.9752
Epoch 9/20
60000/60000 [==============================] - 18s 297us/step - loss:
0.1465 - accuracy: 0.9568 - val_loss: 0.0940 - val_accuracy: 0.9730
Epoch 10/20
60000/60000 [==============================] - 17s 291us/step - loss:
0.1384 - accuracy: 0.9583 - val_loss: 0.0884 - val_accuracy: 0.9736
Epoch 11/20
60000/60000 [==============================] - 18s 293us/step - loss:
0.1333 - accuracy: 0.9604 - val_loss: 0.0882 - val_accuracy: 0.9752
Epoch 12/20
60000/60000 [==============================] - 17s 283us/step - loss:
```

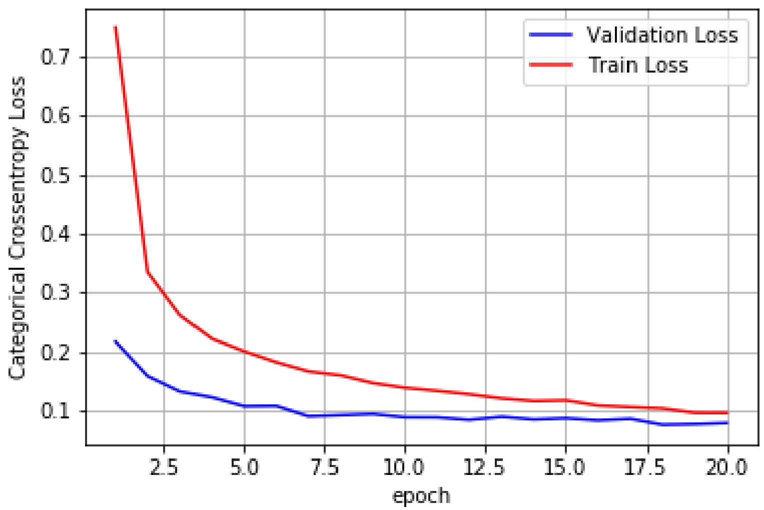
```
0.1275 - accuracy: 0.9625 - val_loss: 0.0840 - val_accuracy: 0.9783

Epoch 13/20
60000/60000 [==============================] - 17s 283us/step - loss:
0.1204 - accuracy: 0.9642 - val_loss: 0.0893 - val_accuracy: 0.9765
Epoch 14/20
60000/60000 [==============================] - 17s 287us/step - loss:
0.1160 - accuracy: 0.9664 - val_loss: 0.0847 - val_accuracy: 0.9751
Epoch 15/20
60000/60000 [==============================] - 17s 284us/step - loss:
0.1171 - accuracy: 0.9654 - val_loss: 0.0867 - val_accuracy: 0.9749
Epoch 16/20
60000/60000 [==============================] - 16s 266us/step - loss:
0.1083 - accuracy: 0.9685 - val_loss: 0.0831 - val_accuracy: 0.9772
Epoch 17/20
60000/60000 [==============================] - 16s 261us/step - loss:
0.1058 - accuracy: 0.9679 - val_loss: 0.0858 - val_accuracy: 0.9763
Epoch 18/20
60000/60000 [==============================] - 17s 285us/step - loss:
0.1033 - accuracy: 0.9694 - val_loss: 0.0757 - val_accuracy: 0.9794
Epoch 19/20
60000/60000 [==============================] - 17s 283us/step - loss:
0.0960 - accuracy: 0.9709 - val_loss: 0.0767 - val_accuracy: 0.9790
Epoch 20/20
60000/60000 [==============================] - 16s 274us/step - loss:
0.0957 - accuracy: 0.9714 - val_loss: 0.0786 - val_accuracy: 0.9780
```

In [43]:

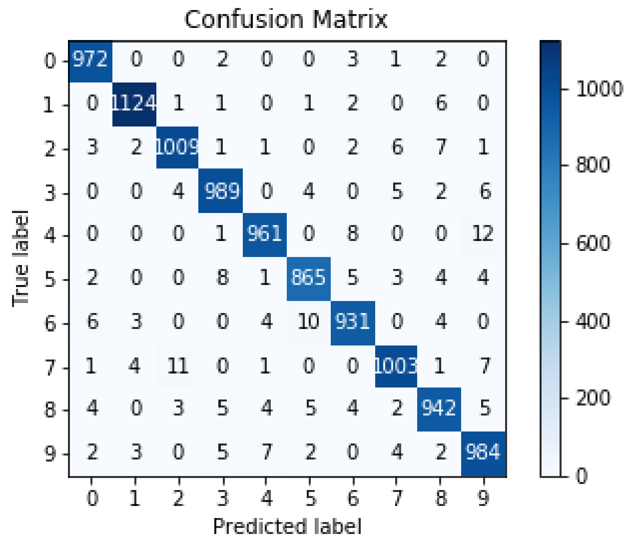
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

first_1 = 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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.07859424627450352
Test accuracy: 0.9779999852180481
```



In [44]:
```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```



Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0xd992bca080>

784-400-250-10 model with batch normalisation

In [ ]:

In [45]:
```
model_drop = Sequential()

model_drop.add(Dense(400, activation='relu', input_shape=(input_dim,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))

model_drop.add(Dense(250, activation='relu', kernel_initializer=RandomN
```

```
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
Model: "sequential_20"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_56 (Dense)             (None, 400)               314000    
_________________________________________________________________
batch_normalization_5 (Batch (None, 400)               1600      
_________________________________________________________________
dense_57 (Dense)             (None, 250)               100250    
_________________________________________________________________
batch_normalization_6 (Batch (None, 250)               1000      
_________________________________________________________________
dense_58 (Dense)             (None, 10)                2510      
=================================================================
Total params: 419,360
Trainable params: 418,060
Non-trainable params: 1,300
_________________________________________________________________
```

In [46]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 24s 407us/step - loss: 0.1960 - accuracy: 0.9411 - val_loss: 0.1154 - val_accuracy: 0.9635
Epoch 2/20
60000/60000 [==============================] - 19s 314us/step - loss:
```

```
0.0745 - accuracy: 0.9775 - val_loss: 0.0919 - val_accuracy: 0.9712
Epoch 3/20
60000/60000 [==============================] - 18s 293us/step - loss:
0.0489 - accuracy: 0.9844 - val_loss: 0.0788 - val_accuracy: 0.9762
Epoch 4/20
60000/60000 [==============================] - 18s 300us/step - loss:
0.0341 - accuracy: 0.9894 - val_loss: 0.0824 - val_accuracy: 0.9740
Epoch 5/20
60000/60000 [==============================] - 20s 325us/step - loss:
0.0255 - accuracy: 0.9920 - val_loss: 0.0854 - val_accuracy: 0.9735
Epoch 6/20
60000/60000 [==============================] - 19s 322us/step - loss:
0.0195 - accuracy: 0.9940 - val_loss: 0.0756 - val_accuracy: 0.9783
Epoch 7/20
60000/60000 [==============================] - 20s 329us/step - loss:
0.0170 - accuracy: 0.9950 - val_loss: 0.0764 - val_accuracy: 0.9770
Epoch 8/20
60000/60000 [==============================] - 19s 317us/step - loss:
0.0157 - accuracy: 0.9948 - val_loss: 0.0916 - val_accuracy: 0.9739
Epoch 9/20
60000/60000 [==============================] - 19s 309us/step - loss:
0.0140 - accuracy: 0.9955 - val_loss: 0.0760 - val_accuracy: 0.9767
Epoch 10/20
60000/60000 [==============================] - 19s 310us/step - loss:
0.0129 - accuracy: 0.9958 - val_loss: 0.0747 - val_accuracy: 0.9785
Epoch 11/20
60000/60000 [==============================] - 19s 319us/step - loss:
0.0102 - accuracy: 0.9967 - val_loss: 0.0728 - val_accuracy: 0.9829
Epoch 12/20
60000/60000 [==============================] - 19s 310us/step - loss:
0.0112 - accuracy: 0.9964 - val_loss: 0.0829 - val_accuracy: 0.9776
Epoch 13/20
60000/60000 [==============================] - 19s 313us/step - loss:
0.0093 - accuracy: 0.9969 - val_loss: 0.0739 - val_accuracy: 0.9823
Epoch 14/20
60000/60000 [==============================] - 18s 306us/step - loss:
0.0103 - accuracy: 0.9965 - val_loss: 0.0773 - val_accuracy: 0.9786
Epoch 15/20
60000/60000 [==============================] - 19s 318us/step - loss:
```

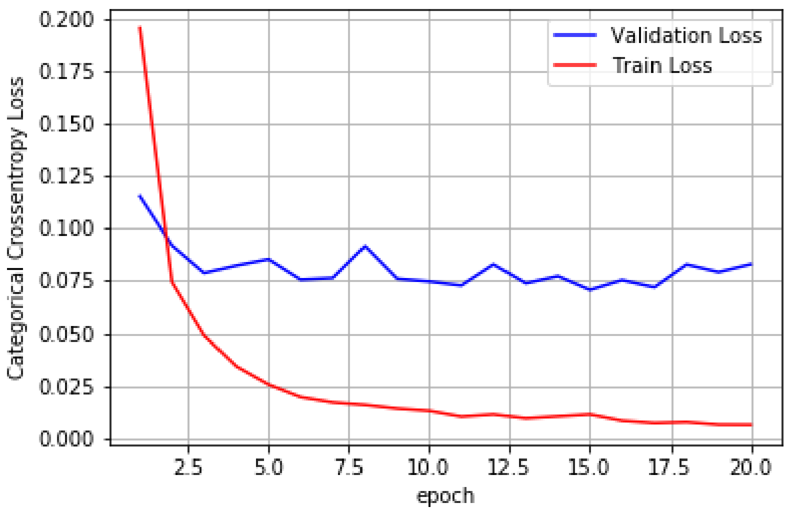
```
0.0112 - accuracy: 0.9962 - val_loss: 0.0707 - val_accuracy: 0.9814
Epoch 16/20
60000/60000 [==============================] - 19s 308us/step - loss:
0.0082 - accuracy: 0.9974 - val_loss: 0.0754 - val_accuracy: 0.9822
Epoch 17/20
60000/60000 [==============================] - 18s 307us/step - loss:
0.0071 - accuracy: 0.9977 - val_loss: 0.0720 - val_accuracy: 0.9807
Epoch 18/20
60000/60000 [==============================] - 19s 313us/step - loss:
0.0075 - accuracy: 0.9978 - val_loss: 0.0829 - val_accuracy: 0.9792
Epoch 19/20
60000/60000 [==============================] - 19s 309us/step - loss:
0.0063 - accuracy: 0.9978 - val_loss: 0.0792 - val_accuracy: 0.9816
Epoch 20/20
60000/60000 [==============================] - 18s 299us/step - loss:
0.0062 - accuracy: 0.9979 - val_loss: 0.0830 - val_accuracy: 0.9792
```

In [47]:

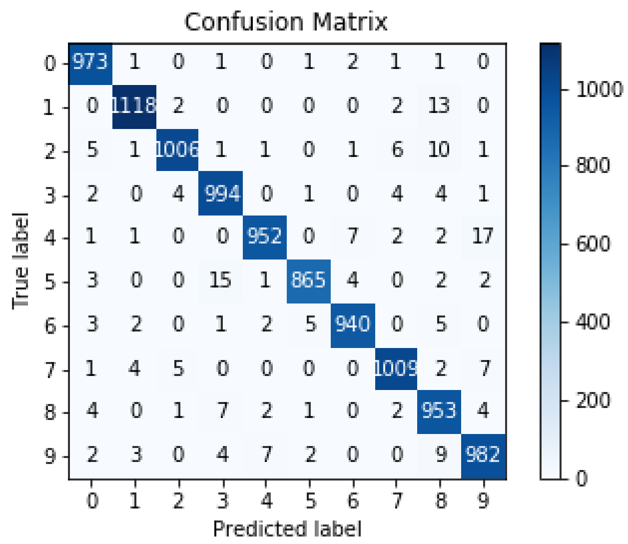
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
first_2 = score[1]

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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.08297608194136047
Test accuracy: 0.9792000005531311
```



In [48]:
```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```



Out[48]:
```
<matplotlib.axes._subplots.AxesSubplot at 0xd9ff7790f0>
```

784-400-250-10 model with batch normalisation and dropout

In [49]:
```
model_drop = Sequential()

model_drop.add(Dense(400, activation='relu', input_shape=(input_dim,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(250, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

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


model_drop.summary()
```

```
Model: "sequential_21"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_59 (Dense)             (None, 400)               314000    
_________________________________________________________________
batch_normalization_7 (Batch (None, 400)               1600      
_________________________________________________________________
dropout_5 (Dropout)          (None, 400)               0         
_________________________________________________________________
dense_60 (Dense)             (None, 250)               100250    
_________________________________________________________________
batch_normalization_8 (Batch (None, 250)               1000      
_________________________________________________________________
dropout_6 (Dropout)          (None, 250)               0         
_________________________________________________________________
dense_61 (Dense)             (None, 10)                2510      
=================================================================
Total params: 419,360
Trainable params: 418,060
Non-trainable params: 1,300
_________________________________________________________________
```

In [50]:

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

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epoch
s=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 26s 426us/step - loss:
0.4870 - accuracy: 0.8523 - val_loss: 0.1786 - val_accuracy: 0.9455
Epoch 2/20
60000/60000 [==============================] - 21s 356us/step - loss:
0.2490 - accuracy: 0.9236 - val_loss: 0.1288 - val_accuracy: 0.9609
Epoch 3/20
60000/60000 [==============================] - 23s 383us/step - loss:
0.2000 - accuracy: 0.9394 - val_loss: 0.1067 - val_accuracy: 0.9669
Epoch 4/20
```

```
Epoch 4/20
60000/60000 [==============================] - 22s 363us/step - loss:
0.1713 - accuracy: 0.9477 - val_loss: 0.0951 - val_accuracy: 0.9696
Epoch 5/20
60000/60000 [==============================] - 21s 356us/step - loss:
0.1507 - accuracy: 0.9537 - val_loss: 0.0939 - val_accuracy: 0.9706
Epoch 6/20
60000/60000 [==============================] - 22s 363us/step - loss:
0.1402 - accuracy: 0.9573 - val_loss: 0.0858 - val_accuracy: 0.9731
Epoch 7/20
60000/60000 [==============================] - 21s 355us/step - loss:
0.1289 - accuracy: 0.9601 - val_loss: 0.0806 - val_accuracy: 0.9761
Epoch 8/20
60000/60000 [==============================] - 22s 360us/step - loss:
0.1202 - accuracy: 0.9633 - val_loss: 0.0750 - val_accuracy: 0.9748
Epoch 9/20
60000/60000 [==============================] - 21s 351us/step - loss:
0.1130 - accuracy: 0.9648 - val_loss: 0.0760 - val_accuracy: 0.9771
Epoch 10/20
60000/60000 [==============================] - 20s 337us/step - loss:
0.1071 - accuracy: 0.9671 - val_loss: 0.0708 - val_accuracy: 0.9788
Epoch 11/20
60000/60000 [==============================] - 21s 347us/step - loss:
0.1034 - accuracy: 0.9677 - val_loss: 0.0728 - val_accuracy: 0.9772
Epoch 12/20
60000/60000 [==============================] - 20s 331us/step - loss:
0.0987 - accuracy: 0.9691 - val_loss: 0.0709 - val_accuracy: 0.9792
Epoch 13/20
60000/60000 [==============================] - 19s 321us/step - loss:
0.0916 - accuracy: 0.9714 - val_loss: 0.0699 - val_accuracy: 0.9787
Epoch 14/20
60000/60000 [==============================] - 21s 342us/step - loss:
0.0879 - accuracy: 0.9722 - val_loss: 0.0667 - val_accuracy: 0.9798
Epoch 15/20
60000/60000 [==============================] - 18s 307us/step - loss:
0.0844 - accuracy: 0.9736 - val_loss: 0.0622 - val_accuracy: 0.9796084
Epoch 16/20
60000/60000 [==============================] - 17s 279us/step - loss:
0.0833 - accuracy: 0.9735 - val_loss: 0.0632 - val_accuracy: 0.9804
Epoch 17/20
```

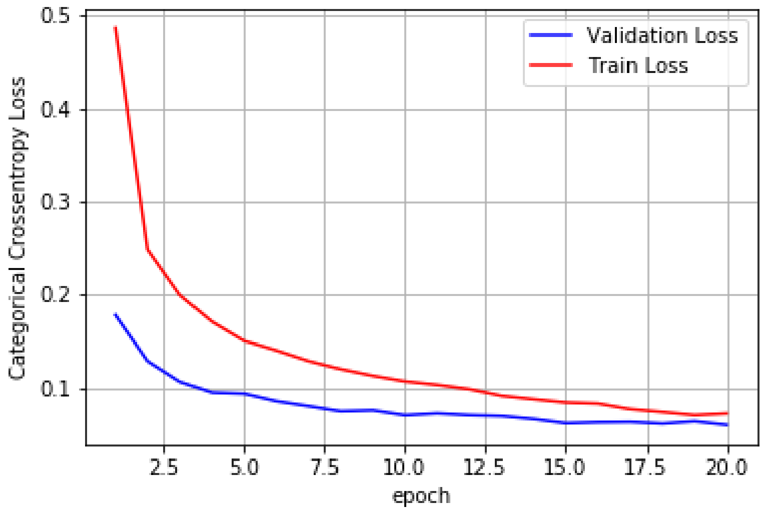
```
60000/60000 [==============================] - 17s 290us/step - loss:
0.0773 - accuracy: 0.9749 - val_loss: 0.0636 - val_accuracy: 0.9807
Epoch 18/20
60000/60000 [==============================] - 17s 280us/step - loss:
0.0742 - accuracy: 0.9764 - val_loss: 0.0618 - val_accuracy: 0.9831
Epoch 19/20
60000/60000 [==============================] - 17s 279us/step - loss:
0.0709 - accuracy: 0.9777 - val_loss: 0.0642 - val_accuracy: 0.9810
Epoch 20/20
60000/60000 [==============================] - 17s 288us/step - loss:
0.0726 - accuracy: 0.9763 - val_loss: 0.0604 - val_accuracy: 0.9823
```

In [51]:

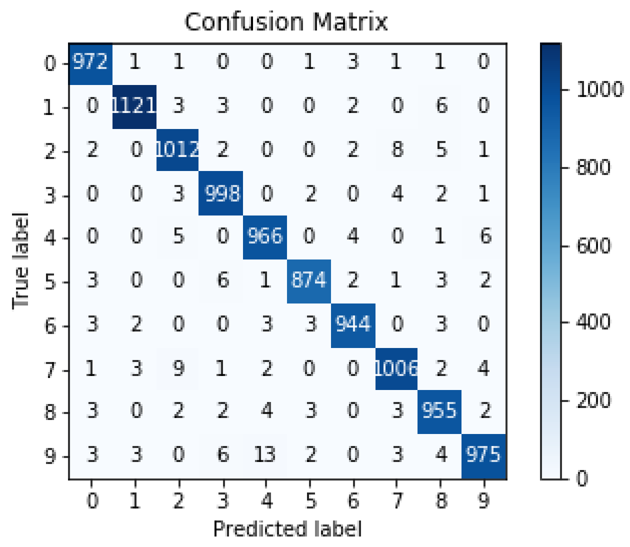
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
first_3 = score[1]

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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.060422330952537594
Test accuracy: 0.9822999835014343
```



In [52]:
```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```



Out[52]: <matplotlib.axes._subplots.AxesSubplot at 0xd98eafb940>

784-600-500-250-10 model with dropout

In [53]:
```
model_drop = Sequential()

model_drop.add(Dense(600, activation='relu', input_shape=(input_dim,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(500, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(250, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))
```

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


model_drop.summary()
```

```
Model: "sequential_22"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_62 (Dense)             (None, 600)               471000    
_________________________________________________________________
dropout_7 (Dropout)          (None, 600)               0         
_________________________________________________________________
dense_63 (Dense)             (None, 500)               300500    
_________________________________________________________________
dropout_8 (Dropout)          (None, 500)               0         
_________________________________________________________________
dense_64 (Dense)             (None, 250)               125250    
_________________________________________________________________
dropout_9 (Dropout)          (None, 250)               0         
_________________________________________________________________
dense_65 (Dense)             (None, 10)                2510      
=================================================================
Total params: 899,260
Trainable params: 899,260
Non-trainable params: 0
_________________________________________________________________
```

In [54]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 25s 411us/step - loss:
```

```
5.6630 - accuracy: 0.3146 - val_loss: 1.2045 - val_accuracy: 0.5896
Epoch 2/20
60000/60000 [==============================] - 23s 376us/step - loss:
1.5286 - accuracy: 0.4616 - val_loss: 1.0148 - val_accuracy: 0.6395
Epoch 3/20
60000/60000 [==============================] - 27s 454us/step - loss:
1.3356 - accuracy: 0.5285 - val_loss: 0.8904 - val_accuracy: 0.6872
Epoch 4/20
60000/60000 [==============================] - 22s 374us/step - loss:
1.1947 - accuracy: 0.5831 - val_loss: 0.7850 - val_accuracy: 0.7325
Epoch 5/20
60000/60000 [==============================] - 23s 382us/step - loss:
1.0543 - accuracy: 0.6477 - val_loss: 0.6306 - val_accuracy: 0.7961
Epoch 6/20
60000/60000 [==============================] - 22s 372us/step - loss:
0.9392 - accuracy: 0.6917 - val_loss: 0.5202 - val_accuracy: 0.8253
Epoch 7/20
60000/60000 [==============================] - 22s 372us/step - loss:
0.8369 - accuracy: 0.7231 - val_loss: 0.4883 - val_accuracy: 0.8341
Epoch 8/20
60000/60000 [==============================] - 23s 376us/step - loss:
0.7448 - accuracy: 0.7541 - val_loss: 0.4659 - val_accuracy: 0.8290
Epoch 9/20
60000/60000 [==============================] - 23s 384us/step - loss:
0.6860 - accuracy: 0.7681 - val_loss: 0.4385 - val_accuracy: 0.8473
Epoch 10/20
60000/60000 [==============================] - 23s 381us/step - loss:
0.6409 - accuracy: 0.7808 - val_loss: 0.4163 - val_accuracy: 0.8515
Epoch 11/20
60000/60000 [==============================] - 23s 376us/step - loss:
0.6130 - accuracy: 0.7902 - val_loss: 0.4156 - val_accuracy: 0.8461
Epoch 12/20
60000/60000 [==============================] - 22s 366us/step - loss:
0.5967 - accuracy: 0.7929 - val_loss: 0.4173 - val_accuracy: 0.8524
Epoch 13/20
60000/60000 [==============================] - 22s 371us/step - loss:
0.5716 - accuracy: 0.8018 - val_loss: 0.3894 - val_accuracy: 0.8607
Epoch 14/20
60000/60000 [==============================] - 22s 367us/step - loss:
```

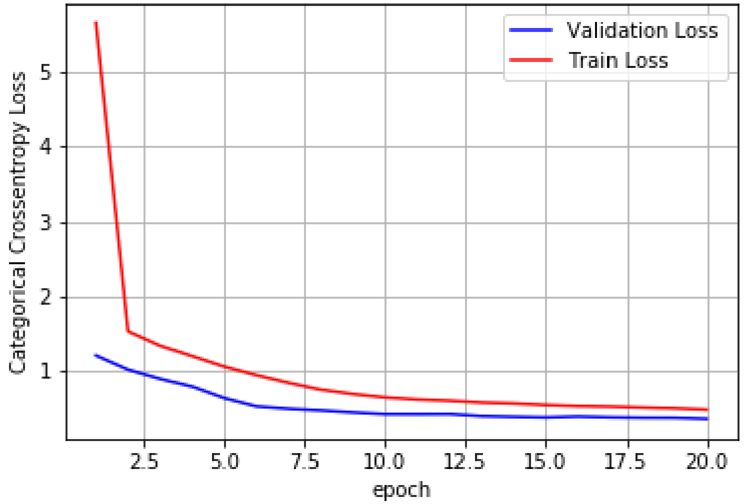
```
0.5602 - accuracy: 0.8034 - val_loss: 0.3800 - val_accuracy: 0.8591
Epoch 15/20
60000/60000 [==============================] - 23s 382us/step - loss:
0.5387 - accuracy: 0.8081 - val_loss: 0.3718 - val_accuracy: 0.8629
Epoch 16/20
60000/60000 [==============================] - 23s 386us/step - loss:
0.5246 - accuracy: 0.8120 - val_loss: 0.3836 - val_accuracy: 0.8601
Epoch 17/20
60000/60000 [==============================] - 22s 374us/step - loss:
0.5176 - accuracy: 0.8146 - val_loss: 0.3733 - val_accuracy: 0.8589
Epoch 18/20
60000/60000 [==============================] - 22s 374us/step - loss:
0.5050 - accuracy: 0.8167 - val_loss: 0.3676 - val_accuracy: 0.8611
Epoch 19/20
60000/60000 [==============================] - 23s 383us/step - loss:
0.4946 - accuracy: 0.8223 - val_loss: 0.3670 - val_accuracy: 0.8668
Epoch 20/20
60000/60000 [==============================] - 23s 378us/step - loss:
0.4770 - accuracy: 0.8257 - val_loss: 0.3531 - val_accuracy: 0.8760
```

In [55]:

```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
second_1 = 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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

# we will get val_loss and val_acc only when you pass the paramter vali
dation_data
```

```
# val_loss : validation loss
# val_acc : validation accuracy

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

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

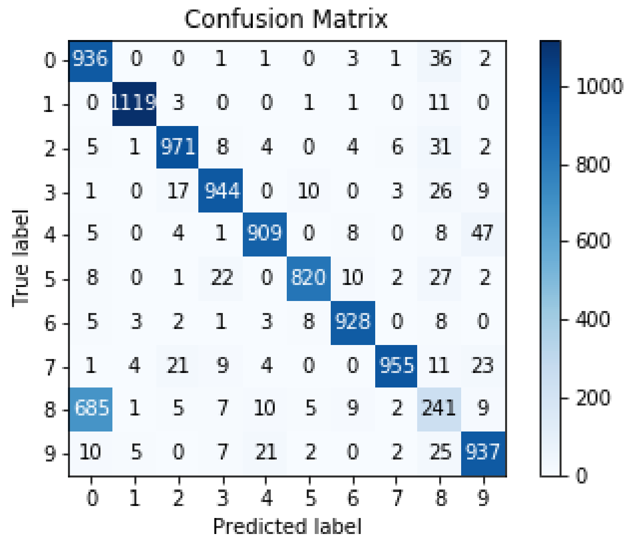
```
Test score: 0.35313249151706694
Test accuracy: 0.8759999871253967
```



In [56]:
```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```

```
C:\Users\hemant\AnacondaNew\lib\site-packages\matplotlib\pyplot.py:514:
```

```
RuntimeWarning: More than 20 figures have been opened. Figures created through the pyplot interface (`matplotlib.pyplot.figure`) are retained until explicitly closed and may consume too much memory. (To control this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



Out[56]: <matplotlib.axes._subplots.AxesSubplot at 0xda2e090278>

784-600-500-250-10 model with batch normalisation

In [57]:

```
model_drop = Sequential()

model_drop.add(Dense(600, activation='relu', input_shape=(input_dim,), kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))

model_drop.add(Dense(500, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
```

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

model_drop.add(Dense(250, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))




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


model_drop.summary()
```

```
Model: "sequential_23"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_66 (Dense)             (None, 600)               471000    
_________________________________________________________________
batch_normalization_9 (Batch (None, 600)               2400      
_________________________________________________________________
dense_67 (Dense)             (None, 500)               300500    
_________________________________________________________________
batch_normalization_10 (Batc (None, 500)               2000      
_________________________________________________________________
dense_68 (Dense)             (None, 250)               125250    
_________________________________________________________________
batch_normalization_11 (Batc (None, 250)               1000      
_________________________________________________________________
dense_69 (Dense)             (None, 10)                2510      
=================================================================
Total params: 904,660
Trainable params: 901,960
Non-trainable params: 2,700
_________________________________________________________________
```

In [58]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', m
```

```
etrics=['accuracy'])

history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epoch
s=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 36s 594us/step - loss:
0.1845 - accuracy: 0.9439 - val_loss: 0.0936 - val_accuracy: 0.97171957
-  - E
Epoch 2/20
60000/60000 [==============================] - 29s 482us/step - loss:
0.0657 - accuracy: 0.9798 - val_loss: 0.0965 - val_accuracy: 0.9687
Epoch 3/20
60000/60000 [==============================] - 29s 482us/step - loss:
0.0431 - accuracy: 0.9869 - val_loss: 0.0801 - val_accuracy: 0.9746
Epoch 4/20
60000/60000 [==============================] - 29s 478us/step - loss:
0.0318 - accuracy: 0.9904 - val_loss: 0.0792 - val_accuracy: 0.9755
Epoch 5/20
60000/60000 [==============================] - 29s 484us/step - loss:
0.0223 - accuracy: 0.9930 - val_loss: 0.0766 - val_accuracy: 0.9757
Epoch 6/20
60000/60000 [==============================] - 29s 487us/step - loss:
0.0189 - accuracy: 0.9938 - val_loss: 0.0818 - val_accuracy: 0.9759l -
ETA: 1s - loss: 0 - ETA: 1s - loss: 0.0 - ETA: 0s - loss: 0.0187 - accu
Epoch 7/20
60000/60000 [==============================] - 29s 484us/step - loss:
0.0205 - accuracy: 0.9934 - val_loss: 0.0875 - val_accuracy: 0.9754
Epoch 8/20
60000/60000 [==============================] - 29s 490us/step - loss:
0.0181 - accuracy: 0.9938 - val_loss: 0.0819 - val_accuracy: 0.9780
Epoch 9/20
60000/60000 [==============================] - 30s 493us/step - loss:
0.0143 - accuracy: 0.9953 - val_loss: 0.0852 - val_accuracy: 0.9780
Epoch 10/20
60000/60000 [==============================] - 30s 492us/step - loss:
0.0143 - accuracy: 0.9950 - val_loss: 0.0886 - val_accuracy: 0.9768
Epoch 11/20
60000/60000 [==============================] - 30s 501us/step - loss:
```

```
0.0131 - accuracy: 0.9954 - val_loss: 0.0767 - val_accuracy: 0.9794

Epoch 12/20
60000/60000 [==============================] - 30s 492us/step - loss:
0.0102 - accuracy: 0.9965 - val_loss: 0.0878 - val_accuracy: 0.9777
Epoch 13/20
60000/60000 [==============================] - 29s 491us/step - loss:
0.0096 - accuracy: 0.9968 - val_loss: 0.0836 - val_accuracy: 0.9787
Epoch 14/20
60000/60000 [==============================] - 29s 489us/step - loss:
0.0099 - accuracy: 0.9967 - val_loss: 0.0777 - val_accuracy: 0.9814
Epoch 15/20
60000/60000 [==============================] - 29s 490us/step - loss:
0.0097 - accuracy: 0.9970 - val_loss: 0.0789 - val_accuracy: 0.9810
Epoch 16/20
60000/60000 [==============================] - 29s 491us/step - loss:
0.0081 - accuracy: 0.9973 - val_loss: 0.0755 - val_accuracy: 0.9809
Epoch 17/20
60000/60000 [==============================] - 30s 507us/step - loss:
0.0092 - accuracy: 0.9972 - val_loss: 0.0729 - val_accuracy: 0.9825
Epoch 18/20
60000/60000 [==============================] - 31s 517us/step - loss:
0.0079 - accuracy: 0.9971 - val_loss: 0.0835 - val_accuracy: 0.9797
Epoch 19/20
60000/60000 [==============================] - 32s 535us/step - loss:
0.0095 - accuracy: 0.9969 - val_loss: 0.0752 - val_accuracy: 0.9814
Epoch 20/20
60000/60000 [==============================] - 34s 561us/step - loss:
0.0050 - accuracy: 0.9984 - val_loss: 0.0715 - val_accuracy: 0.9831ss:
0.0049 -
```

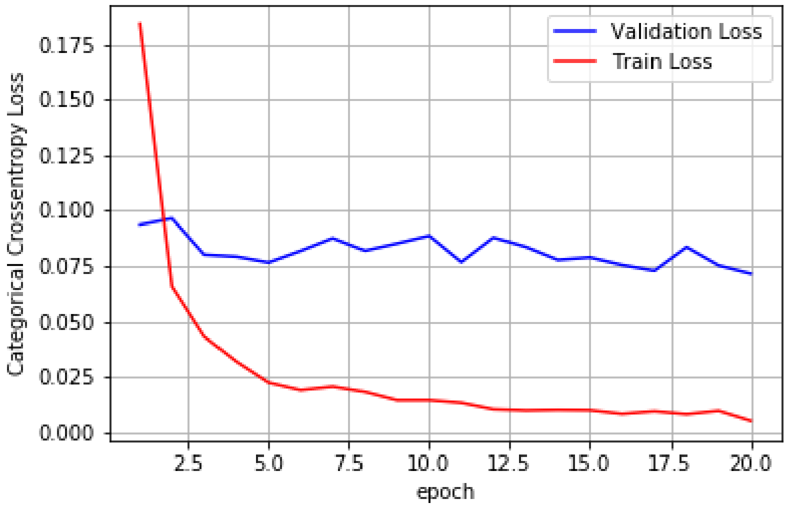
In [59]:

```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
second_2 = 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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

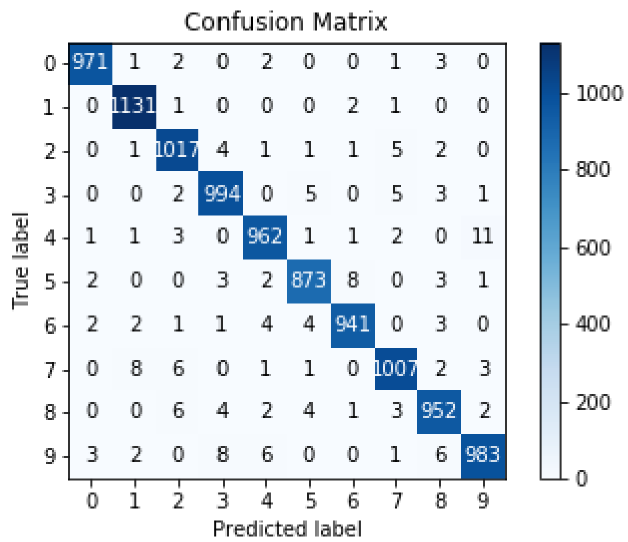
```
Test score: 0.07154271629433848
Test accuracy: 0.9830999970436096
```

```
C:\Users\hemant\AnacondaNew\lib\site-packages\matplotlib\pyplot.py:514:
RuntimeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retained
until explicitly closed and may consume too much memory. (To control th
is warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



In [60]:

```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```



Out[60]: <matplotlib.axes._subplots.AxesSubplot at 0xda320dc860>

784-600-500-250-10 model with batch normalisation and dropout

In [61]:
```
model_drop = Sequential()

model_drop.add(Dense(600, activation='relu', input_shape=(input_dim,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(500, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(250, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))
```

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


model_drop.summary()
```

```
Model: "sequential_24"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_70 (Dense)             (None, 600)               471000
_________________________________________________________________
batch_normalization_12 (Batc (None, 600)               2400
_________________________________________________________________
dropout_10 (Dropout)         (None, 600)               0
_________________________________________________________________
dense_71 (Dense)             (None, 500)               300500
_________________________________________________________________
batch_normalization_13 (Batc (None, 500)               2000
_________________________________________________________________
dropout_11 (Dropout)         (None, 500)               0
_________________________________________________________________
dense_72 (Dense)             (None, 250)               125250
_________________________________________________________________
batch_normalization_14 (Batc (None, 250)               1000
_________________________________________________________________
dropout_12 (Dropout)         (None, 250)               0
_________________________________________________________________
dense_73 (Dense)             (None, 10)                2510
=================================================================
Total params: 904,660
Trainable params: 901,960
Non-trainable params: 2,700
_________________________________________________________________
```

In [62]:
```python
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
history = model_drop.fit(X_train, Y_train, batch_size=batch_size, epoch
s=nb_epoch, verbose=1, validation_data=(X_test, Y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 57s 955us/step - loss:
0.6734 - accuracy: 0.7911 - val_loss: 0.2105 - val_accuracy: 0.9344
Epoch 2/20
60000/60000 [==============================] - 44s 726us/step - loss:
0.3046 - accuracy: 0.9082 - val_loss: 0.1509 - val_accuracy: 0.9522A: 2
6s - loss: 0.3353 - accuracy: 0 - ETA: 26s - loss: 0.3340 - accuracy:
0.8 - ETA: 26s - loss: 0.3323 - accuracy: 0.899 - ETA: 25s - loss: 0.33
20 - accuracy:  - ETA: 25s - loss: 0. - ETA: 23s - loss: 0.33 - ETA: 21
s - loss: 0.3299 - acc - ETA: 20s - loss: 0.3289 - accurac - ETA: 4s -
loss:
Epoch 3/20
60000/60000 [==============================] - 47s 777us/step - loss:
0.2350 - accuracy: 0.9296 - val_loss: 0.1232 - val_accuracy: 0.9610
Epoch 4/20
60000/60000 [==============================] - 47s 777us/step - loss:
0.1995 - accuracy: 0.9401 - val_loss: 0.1096 - val_accuracy: 0.9657
Epoch 5/20
60000/60000 [==============================] - 43s 718us/step - loss:
0.1757 - accuracy: 0.9460 - val_loss: 0.0962 - val_accuracy: 0.9705
Epoch 6/20
60000/60000 [==============================] - 36s 599us/step - loss:
0.1586 - accuracy: 0.9521 - val_loss: 0.0884 - val_accuracy: 0.9720
Epoch 7/20
60000/60000 [==============================] - 34s 561us/step - loss:
0.1459 - accuracy: 0.9555 - val_loss: 0.0830 - val_accuracy: 0.9743
Epoch 8/20
60000/60000 [==============================] - 34s 572us/step - loss:
0.1339 - accuracy: 0.9589 - val_loss: 0.0863 - val_accuracy: 0.9723
Epoch 9/20
60000/60000 [==============================] - 35s 590us/step - loss:
0.1233 - accuracy: 0.9625 - val_loss: 0.0801 - val_accuracy: 0.9738
Epoch 10/20
60000/60000 [==============================] - 36s 592us/step - loss:
0.1150 - accuracy: 0.9642 - val_loss: 0.0764 - val_accuracy: 0.9762
```

```
Epoch 11/20
60000/60000 [==============================] - 34s 571us/step - loss:
0.1071 - accuracy: 0.9676 - val_loss: 0.0756 - val_accuracy: 0.9786
Epoch 12/20
60000/60000 [==============================] - 33s 543us/step - loss:
0.1043 - accuracy: 0.9679 - val_loss: 0.0710 - val_accuracy: 0.9793
Epoch 13/20
60000/60000 [==============================] - 34s 563us/step - loss:
0.0995 - accuracy: 0.9694 - val_loss: 0.0705 - val_accuracy: 0.9786
Epoch 14/20
60000/60000 [==============================] - 35s 580us/step - loss:
0.0904 - accuracy: 0.9718 - val_loss: 0.0720 - val_accuracy: 0.9785
Epoch 15/20
60000/60000 [==============================] - 35s 584us/step - loss:
0.0874 - accuracy: 0.9729 - val_loss: 0.0645 - val_accuracy: 0.9795
Epoch 16/20
60000/60000 [==============================] - 35s 584us/step - loss:
0.0856 - accuracy: 0.9743 - val_loss: 0.0612 - val_accuracy: 0.9816
Epoch 17/20
60000/60000 [==============================] - 35s 584us/step - loss:
0.0822 - accuracy: 0.9740 - val_loss: 0.0608 - val_accuracy: 0.9806
Epoch 18/20
60000/60000 [==============================] - 36s 593us/step - loss:
0.0780 - accuracy: 0.9758 - val_loss: 0.0641 - val_accuracy: 0.9828
Epoch 19/20
60000/60000 [==============================] - 35s 581us/step - loss:
0.0749 - accuracy: 0.9762 - val_loss: 0.0683 - val_accuracy: 0.9798
Epoch 20/20
60000/60000 [==============================] - 34s 572us/step - loss:
0.0715 - accuracy: 0.9777 - val_loss: 0.0604 - val_accuracy: 0.9822
```

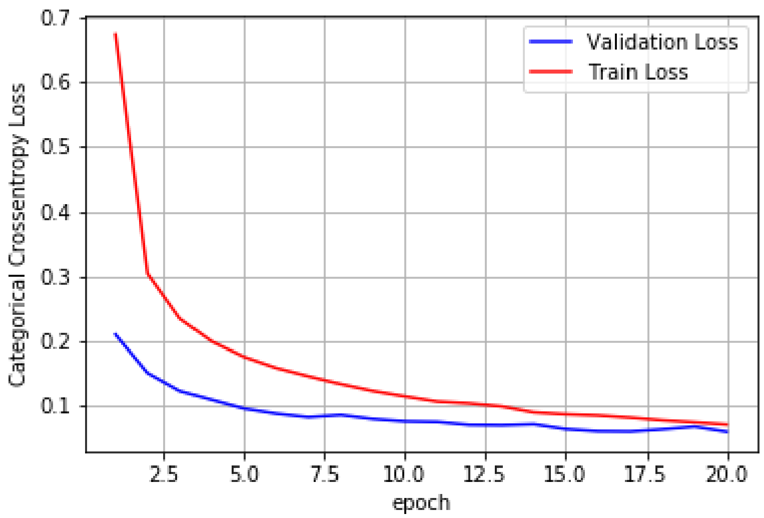
In [63]:

```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
second_3 = 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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

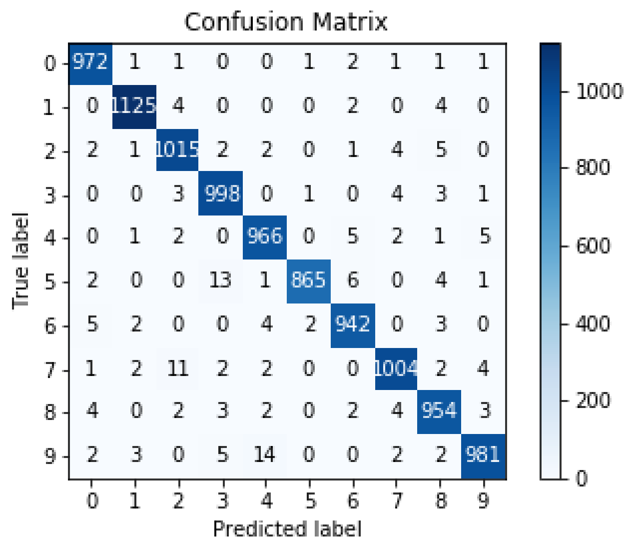
```
Test score: 0.06040368790557841
Test accuracy: 0.982200026512146
```

```
C:\Users\hemant\AnacondaNew\lib\site-packages\matplotlib\pyplot.py:514:
RuntimeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retained
until explicitly closed and may consume too much memory. (To control th
is warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



In [64]:

```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```



Out[64]: <matplotlib.axes._subplots.AxesSubplot at 0xda4efc3fd0>

784-600-500-400-300-200-10 model with dropout

In [65]:
```
model_drop = Sequential()

model_drop.add(Dense(600, activation='relu', input_shape=(input_dim,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(500, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(400, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))
```

```
model_drop.add(Dense(300, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(200, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
#model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))




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


model_drop.summary()
```

```
Model: "sequential_25"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_74 (Dense)             (None, 600)               471000    
_________________________________________________________________
dropout_13 (Dropout)         (None, 600)               0         
_________________________________________________________________
dense_75 (Dense)             (None, 500)               300500    
_________________________________________________________________
dropout_14 (Dropout)         (None, 500)               0         
_________________________________________________________________
dense_76 (Dense)             (None, 400)               200400    
_________________________________________________________________
dropout_15 (Dropout)         (None, 400)               0         
_________________________________________________________________
dense_77 (Dense)             (None, 300)               120300    
_________________________________________________________________
dropout_16 (Dropout)         (None, 300)               0         
_________________________________________________________________
dense_78 (Dense)             (None, 200)               60200     
_________________________________________________________________
```

```
dropout_17 (Dropout)         (None, 200)               0
_________________________________________________________________
dense_79 (Dense)             (None, 10)                2010
=================================================================
Total params: 1,154,410
Trainable params: 1,154,410
Non-trainable params: 0
_________________________________________________________________
```

In [66]:
```
model_drop.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

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

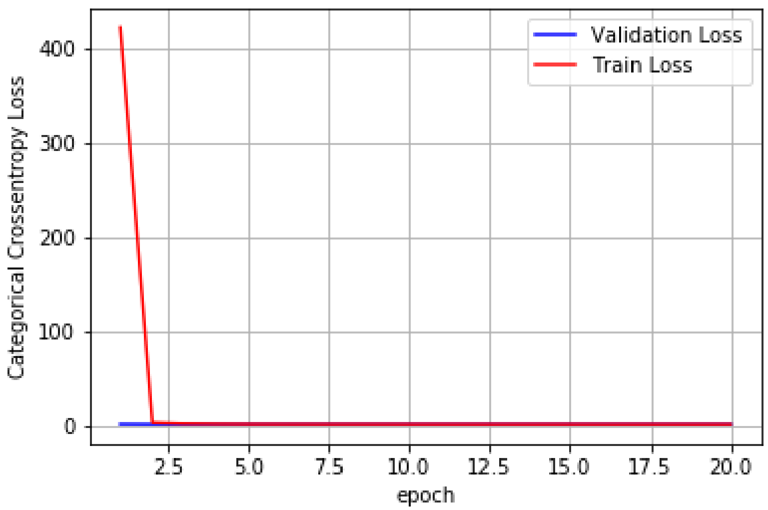
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 31s 516us/step - loss: 422.5219 - accuracy: 0.1038 - val_loss: 2.4589 - val_accuracy: 0.1124 1s - loss: 437
Epoch 2/20
60000/60000 [==============================] - 28s 467us/step - loss: 4.1316 - accuracy: 0.1110 - val_loss: 2.3373 - val_accuracy: 0.1148
Epoch 3/20
60000/60000 [==============================] - 34s 559us/step - loss: 2.9623 - accuracy: 0.1119 - val_loss: 2.3235 - val_accuracy: 0.1137
Epoch 4/20
60000/60000 [==============================] - 33s 546us/step - loss: 2.5956 - accuracy: 0.1123 - val_loss: 2.3179 - val_accuracy: 0.1134
Epoch 5/20
60000/60000 [==============================] - 31s 522us/step - loss: 2.4333 - accuracy: 0.1123 - val_loss: 2.3142 - val_accuracy: 0.1137
Epoch 6/20
60000/60000 [==============================] - 31s 522us/step - loss: 2.3887 - accuracy: 0.1125 - val_loss: 2.3200 - val_accuracy: 0.1137
Epoch 7/20
60000/60000 [==============================] - 27s 448us/step - loss: 2.3541 - accuracy: 0.1123 - val_loss: 2.3114 - val_accuracy: 0.1135
Epoch 8/20
60000/60000 [==============================] - 26s 436us/step - loss:
```

```
2.3543 - accuracy: 0.1124 - val_loss: 2.3080 - val_accuracy: 0.1134
Epoch 9/20
60000/60000 [==============================] - 27s 454us/step - loss:
2.3382 - accuracy: 0.1124 - val_loss: 2.3071 - val_accuracy: 0.1137
Epoch 10/20
60000/60000 [==============================] - 30s 505us/step - loss:
2.3233 - accuracy: 0.1124 - val_loss: 2.3077 - val_accuracy: 0.1137
Epoch 11/20
60000/60000 [==============================] - 31s 519us/step - loss:
2.3156 - accuracy: 0.1123 - val_loss: 2.3069 - val_accuracy: 0.1139
Epoch 12/20
60000/60000 [==============================] - 32s 532us/step - loss:
2.3108 - accuracy: 0.1123 - val_loss: 2.3075 - val_accuracy: 0.1134
Epoch 13/20
60000/60000 [==============================] - 29s 489us/step - loss:
2.3135 - accuracy: 0.1124 - val_loss: 2.3072 - val_accuracy: 0.1134
Epoch 14/20
60000/60000 [==============================] - 27s 455us/step - loss:
2.3094 - accuracy: 0.1123 - val_loss: 2.3080 - val_accuracy: 0.1133
Epoch 15/20
60000/60000 [==============================] - 32s 530us/step - loss:
2.3208 - accuracy: 0.1124 - val_loss: 2.3072 - val_accuracy: 0.1134
Epoch 16/20
60000/60000 [==============================] - 32s 528us/step - loss:
2.3013 - accuracy: 0.1124 - val_loss: 2.3072 - val_accuracy: 0.1135
Epoch 17/20
60000/60000 [==============================] - 28s 472us/step - loss:
2.3013 - accuracy: 0.1124 - val_loss: 2.3073 - val_accuracy: 0.1135
Epoch 18/20
60000/60000 [==============================] - 30s 507us/step - loss:
2.3013 - accuracy: 0.1124 - val_loss: 2.3073 - val_accuracy: 0.1135
Epoch 19/20
60000/60000 [==============================] - 32s 534us/step - loss:
2.3012 - accuracy: 0.1124 - val_loss: 2.3072 - val_accuracy: 0.1135
Epoch 20/20
60000/60000 [==============================] - 32s 534us/step - loss:
2.3012 - accuracy: 0.1124 - val_loss: 2.3073 - val_accuracy: 0.1135
```

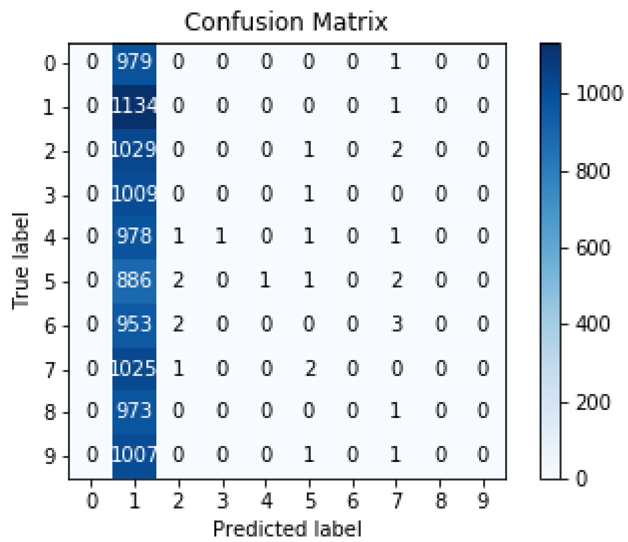
In [67]:
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
third_1 = score[1]

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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 2.307284020996094
Test accuracy: 0.11349999904632568
```

```
C:\Users\hemant\AnacondaNew\lib\site-packages\matplotlib\pyplot.py:514:
RuntimeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retained
until explicitly closed and may consume too much memory. (To control th
is warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



In [68]:
```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```



Out[68]: <matplotlib.axes._subplots.AxesSubplot at 0xd98ea8b6a0>

784-600-500-400-300-200-10 model with batch normalisation

In [69]:
```
model_drop = Sequential()

model_drop.add(Dense(600, activation='relu', input_shape=(input_dim,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))

model_drop.add(Dense(500, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))

model_drop.add(Dense(400, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))
```

```
model_drop.add(Dense(300, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))

model_drop.add(Dense(200, activation='relu', kernel_initializer=RandomNormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
#model_drop.add(Dropout(0.5))




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


model_drop.summary()
```

```
Model: "sequential_26"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_80 (Dense)             (None, 600)               471000
_________________________________________________________________
batch_normalization_15 (Batc (None, 600)               2400
_________________________________________________________________
dense_81 (Dense)             (None, 500)               300500
_________________________________________________________________
batch_normalization_16 (Batc (None, 500)               2000
_________________________________________________________________
dense_82 (Dense)             (None, 400)               200400
_________________________________________________________________
batch_normalization_17 (Batc (None, 400)               1600
_________________________________________________________________
dense_83 (Dense)             (None, 300)               120300
_________________________________________________________________
batch_normalization_18 (Batc (None, 300)               1200
_________________________________________________________________
dense_84 (Dense)             (None, 200)               60200
_________________________________________________________________
```

```
batch_normalization_19 (Batc (None, 200)               800
_________________________________________________________________
dense_85 (Dense)             (None, 10)                2010
=================================================================
Total params: 1,162,410
Trainable params: 1,158,410
Non-trainable params: 4,000
_________________________________________________________________
```

In [70]:

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

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 55s 924us/step - loss: 0.2320 - accuracy: 0.9299 - val_loss: 0.1257 - val_accuracy: 0.9596
Epoch 2/20
60000/60000 [==============================] - 51s 845us/step - loss: 0.0789 - accuracy: 0.9758 - val_loss: 0.1042 - val_accuracy: 0.9680
Epoch 3/20
60000/60000 [==============================] - 51s 843us/step - loss: 0.0524 - accuracy: 0.9834 - val_loss: 0.0817 - val_accuracy: 0.9752
Epoch 4/20
60000/60000 [==============================] - 52s 869us/step - loss: 0.0384 - accuracy: 0.9874 - val_loss: 0.0959 - val_accuracy: 0.9709
Epoch 5/20
60000/60000 [==============================] - 49s 808us/step - loss: 0.0335 - accuracy: 0.9887 - val_loss: 0.0768 - val_accuracy: 0.9776
Epoch 6/20
60000/60000 [==============================] - 48s 794us/step - loss: 0.0258 - accuracy: 0.9915 - val_loss: 0.0840 - val_accuracy: 0.9757
Epoch 7/20
60000/60000 [==============================] - 47s 785us/step - loss: 0.0230 - accuracy: 0.9925 - val_loss: 0.0909 - val_accuracy: 0.9731
Epoch 8/20
60000/60000 [==============================] - 44s 736us/step - loss: 0.0226 - accuracy: 0.9923 - val_loss: 0.1015 - val_accuracy: 0.9724
```

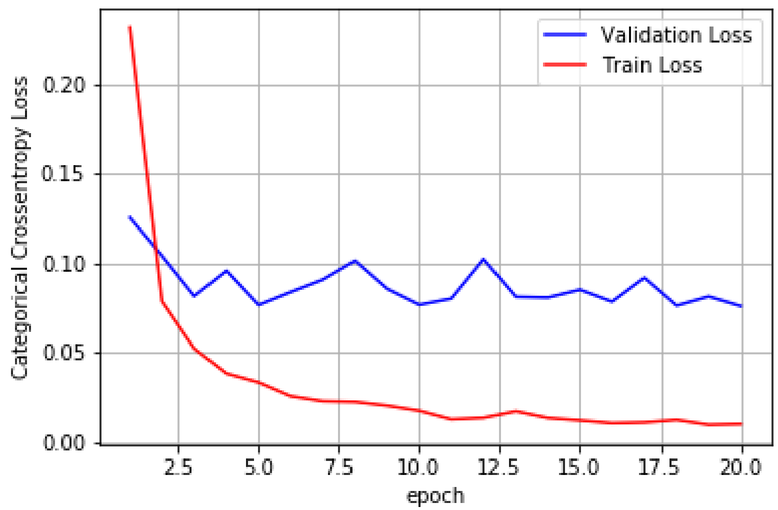
```
Epoch 9/20
60000/60000 [==============================] - 41s 679us/step - loss:
0.0205 - accuracy: 0.9931 - val_loss: 0.0858 - val_accuracy: 0.9774
Epoch 10/20
60000/60000 [==============================] - 39s 658us/step - loss:
0.0176 - accuracy: 0.9941 - val_loss: 0.0769 - val_accuracy: 0.9795
Epoch 11/20
60000/60000 [==============================] - 46s 768us/step - loss:
0.0128 - accuracy: 0.9955 - val_loss: 0.0804 - val_accuracy: 0.9791
Epoch 12/20
60000/60000 [==============================] - 47s 786us/step - loss:
0.0137 - accuracy: 0.9954 - val_loss: 0.1024 - val_accuracy: 0.9731
Epoch 13/20
60000/60000 [==============================] - 48s 800us/step - loss:
0.0173 - accuracy: 0.9945 - val_loss: 0.0814 - val_accuracy: 0.9786
Epoch 14/20
60000/60000 [==============================] - 48s 807us/step - loss:
0.0136 - accuracy: 0.9952 - val_loss: 0.0810 - val_accuracy: 0.9798
Epoch 15/20
60000/60000 [==============================] - 49s 819us/step - loss:
0.0122 - accuracy: 0.9956 - val_loss: 0.0853 - val_accuracy: 0.9783
Epoch 16/20
60000/60000 [==============================] - 49s 824us/step - loss:
0.0107 - accuracy: 0.9964 - val_loss: 0.0786 - val_accuracy: 0.9789
Epoch 17/20
60000/60000 [==============================] - 50s 826us/step - loss:
0.0111 - accuracy: 0.9961 - val_loss: 0.0920 - val_accuracy: 0.9781
Epoch 18/20
60000/60000 [==============================] - 50s 840us/step - loss:
0.0125 - accuracy: 0.9957 - val_loss: 0.0764 - val_accuracy: 0.9815
Epoch 19/20
60000/60000 [==============================] - 49s 822us/step - loss:
0.0098 - accuracy: 0.9965 - val_loss: 0.0815 - val_accuracy: 0.9807
Epoch 20/20
60000/60000 [==============================] - 50s 831us/step - loss:
0.0102 - accuracy: 0.9966 - val_loss: 0.0762 - val_accuracy: 0.9821
```

In [71]:
```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
```

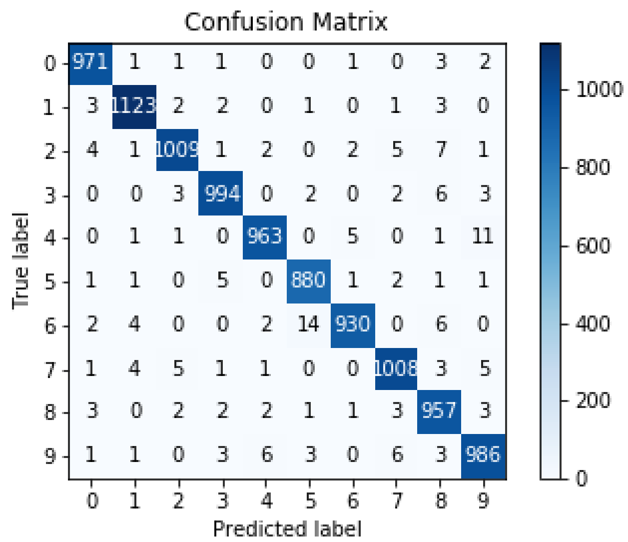
```
print('Test score:', score[0])
print('Test accuracy:', score[1])
third_2 = score[1]

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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.07619471673628853
Test accuracy: 0.9821000099182129
```

```
C:\Users\hemant\AnacondaNew\lib\site-packages\matplotlib\pyplot.py:514:
RuntimeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retained
until explicitly closed and may consume too much memory. (To control th
is warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



In [72]:
```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```



Out[72]: <matplotlib.axes._subplots.AxesSubplot at 0xda87edccc0>

784-600-500-400-300-200-10 model with batch normalisation and dropout

In [73]:
```
model_drop = Sequential()

model_drop.add(Dense(600, activation='relu', input_shape=(input_dim,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.039, seed=None)))
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(500, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(400, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))
```

```
model_drop.add(Dense(300, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))

model_drop.add(Dense(200, activation='relu', kernel_initializer=RandomN
ormal(mean=0.0, stddev=0.55, seed=None)) )
model_drop.add(BatchNormalization())
model_drop.add(Dropout(0.5))




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


model_drop.summary()
```

```
Model: "sequential_27"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_86 (Dense)             (None, 600)               471000    
_________________________________________________________________
batch_normalization_20 (Batc (None, 600)               2400      
_________________________________________________________________
dropout_18 (Dropout)         (None, 600)               0         
_________________________________________________________________
dense_87 (Dense)             (None, 500)               300500    
_________________________________________________________________
batch_normalization_21 (Batc (None, 500)               2000      
_________________________________________________________________
dropout_19 (Dropout)         (None, 500)               0         
_________________________________________________________________
dense_88 (Dense)             (None, 400)               200400    
_________________________________________________________________
batch_normalization_22 (Batc (None, 400)               1600      
_________________________________________________________________
dropout_20 (Dropout)         (None, 400)               0         
_________________________________________________________________
```

```
dense_89 (Dense)             (None, 300)               120300
_________________________________________________________________
batch_normalization_23 (Batc (None, 300)               1200
_________________________________________________________________
dropout_21 (Dropout)         (None, 300)               0
_________________________________________________________________
dense_90 (Dense)             (None, 200)               60200
_________________________________________________________________
batch_normalization_24 (Batc (None, 200)               800
_________________________________________________________________
dropout_22 (Dropout)         (None, 200)               0
_________________________________________________________________
dense_91 (Dense)             (None, 10)                2010
=================================================================
Total params: 1,162,410
Trainable params: 1,158,410
Non-trainable params: 4,000
_________________________________________________________________
```

In [74]:

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

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

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 68s 1ms/step - loss: 1.4857 - accuracy: 0.5218 - val_loss: 0.4887 - val_accuracy: 0.8527
Epoch 2/20
60000/60000 [==============================] - 61s 1ms/step - loss: 0.5621 - accuracy: 0.8240 - val_loss: 0.2641 - val_accuracy: 0.9221
Epoch 3/20
60000/60000 [==============================] - 60s 998us/step - loss: 0.3946 - accuracy: 0.8819 - val_loss: 0.2052 - val_accuracy: 0.9384
Epoch 4/20
60000/60000 [==============================] - 58s 967us/step - loss: 0.3143 - accuracy: 0.9079 - val_loss: 0.1713 - val_accuracy: 0.9498
Epoch 5/20
60000/60000 [==============================] - 57s 948us/step - loss:
```

```
0.2652 - accuracy: 0.9219 - val_loss: 0.1543 - val_accuracy: 0.9558
Epoch 6/20
60000/60000 [==============================] - 57s 952us/step - loss:
0.2345 - accuracy: 0.9324 - val_loss: 0.1423 - val_accuracy: 0.9594
Epoch 7/20
60000/60000 [==============================] - 57s 950us/step - loss:
0.2126 - accuracy: 0.9385 - val_loss: 0.1192 - val_accuracy: 0.9655
Epoch 8/20
60000/60000 [==============================] - 57s 958us/step - loss:
0.1903 - accuracy: 0.9450 - val_loss: 0.1135 - val_accuracy: 0.9677
Epoch 9/20
60000/60000 [==============================] - 57s 946us/step - loss:
0.1774 - accuracy: 0.9485 - val_loss: 0.0970 - val_accuracy: 0.9736
Epoch 10/20
60000/60000 [==============================] - 57s 951us/step - loss:
0.1660 - accuracy: 0.9528 - val_loss: 0.0927 - val_accuracy: 0.9735
Epoch 11/20
60000/60000 [==============================] - 56s 939us/step - loss:
0.1617 - accuracy: 0.9541 - val_loss: 0.0900 - val_accuracy: 0.9745
Epoch 12/20
60000/60000 [==============================] - 49s 821us/step - loss:
0.1514 - accuracy: 0.9568 - val_loss: 0.0855 - val_accuracy: 0.9775
Epoch 13/20
60000/60000 [==============================] - 49s 823us/step - loss:
0.1393 - accuracy: 0.9607 - val_loss: 0.0866 - val_accuracy: 0.9761
Epoch 14/20
60000/60000 [==============================] - 54s 898us/step - loss:
0.1320 - accuracy: 0.9617 - val_loss: 0.0803 - val_accuracy: 0.9783
Epoch 15/20
60000/60000 [==============================] - 59s 985us/step - loss:
0.1272 - accuracy: 0.9634 - val_loss: 0.0861 - val_accuracy: 0.9785
Epoch 16/20
60000/60000 [==============================] - 59s 977us/step - loss:
0.1256 - accuracy: 0.9646 - val_loss: 0.0827 - val_accuracy: 0.9780
Epoch 17/20
60000/60000 [==============================] - 59s 985us/step - loss:
0.1169 - accuracy: 0.9671 - val_loss: 0.0827 - val_accuracy: 0.9782
Epoch 18/20
60000/60000 [==============================] - 60s 997us/step - loss:
```

```
0.1120 - accuracy: 0.9671 - val_loss: 0.0788 - val_accuracy: 0.9795
Epoch 19/20
60000/60000 [==============================] - 60s 998us/step - loss:
0.1073 - accuracy: 0.9692 - val_loss: 0.0777 - val_accuracy: 0.9805
Epoch 20/20
60000/60000 [==============================] - 60s 998us/step - loss:
0.1051 - accuracy: 0.9700 - val_loss: 0.0738 - val_accuracy: 0.9815
```

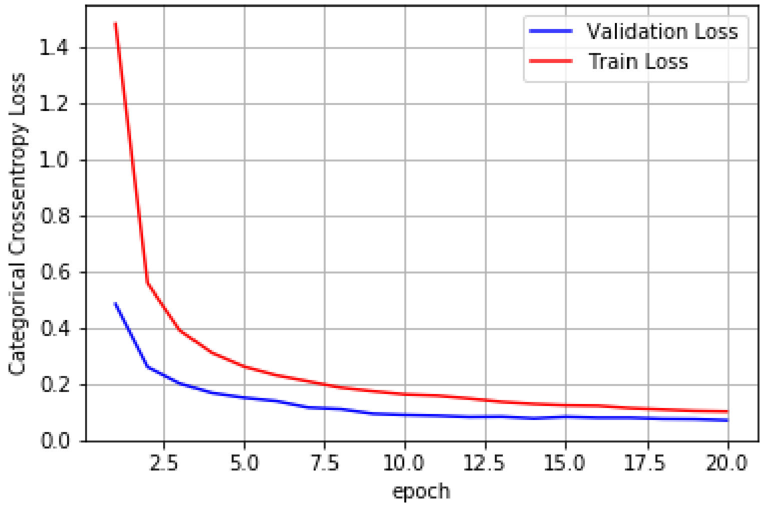
In [75]:

```
score = model_drop.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
third_3 = 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, epo
chs=nb_epoch, verbose=1, validation_data=(X_test, Y_test))

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

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

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

```
Test score: 0.07378138729266356
```

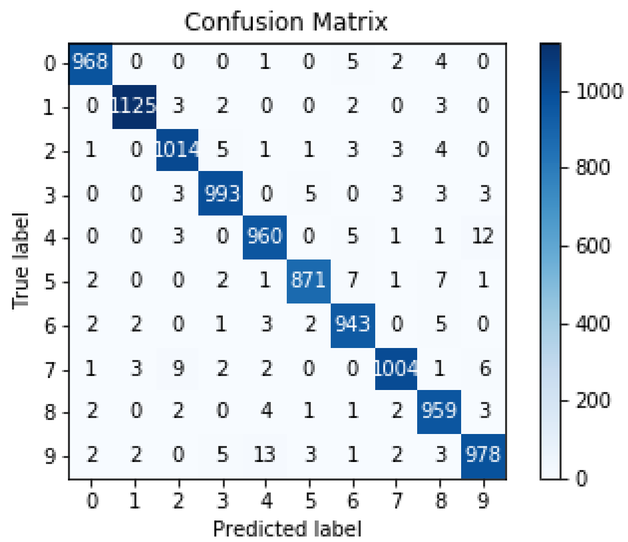
```
Test accuracy: 0.9815000295639038
```

```
C:\Users\hemant\AnacondaNew\lib\site-packages\matplotlib\pyplot.py:514:
RuntimeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retained
until explicitly closed and may consume too much memory. (To control th
is warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



In [76]:

```
pred=model_drop.predict(X_test)
#pred= (pred>0.5)
import scikitplot.metrics as skplt
skplt.plot_confusion_matrix(Y_test.argmax(axis=1), pred.argmax(axis=1))
```



Out[76]: <matplotlib.axes._subplots.AxesSubplot at 0xdaba2489b0>

In [78]:

```
from prettytable import PrettyTable
x = PrettyTable()
x.field_names = ["model","accuracy"]
x.add_row(["784-400-250-10 model with dropout",first_1])
x.add_row(["784-400-250-10 model with batch normalisation",first_2])
x.add_row(["784-400-250-10 model with batch normalisation and dropout",
first_3])
x.add_row(["784-600-500-250-10 model with dropout",second_1])

x.add_row(["784-600-500-250-10 model with batch normalisation",second_2
])
x.add_row(["784-600-500-250-10 model with batch normalisation and dropo
ut",second_3])
x.add_row(["784-600-500-400-300-200-10 model with dropout",third_1])
x.add_row(["784-600-500-400-300-200-10 model with batch normalisation",
third_2])
x.add_row(["784-600-500-400-300-200-10 model with batch normalisation a
nd dropout",third_3])
```

In [79]:
```
print(x)
```

```
+-----------------------------------------------------------------------
-+--------------------+
|                                 model
|       accuracy      |
+-----------------------------------------------------------------------
-+--------------------+
|                   784-400-250-10 model with dropout
|  0.9779999852180481 |
|             784-400-250-10 model with batch normalisation
|  0.979200005531311  |
|       784-400-250-10 model with batch normalisation and dropout
|  0.9822999835014343 |
|                 784-600-500-250-10 model with dropout
|  0.8759999871253967 |
|           784-600-500-250-10 model with batch normalisation
|  0.9830999970436096 |
|     784-600-500-250-10 model with batch normalisation and dropout
|  0.982200026512146  |
|             784-600-500-400-300-200-10 model with dropout
| 0.11349999904632568 |
|       784-600-500-400-300-200-10 model with batch normalisation
|  0.9821000099182129 |
| 784-600-500-400-300-200-10 model with batch normalisation and dropout
|  0.9815000295639038 |
+-----------------------------------------------------------------------
-+--------------------+
```

accordingly above table,"784-600-500-250-10 model with batch normalisation" have highest accuracy. so we will use this model.

In [ ]: